Analysis of Algorithms Project

Team Members:

Venkata Sai Karthik Metlapalli - 65764476 -> Task 3B, Task 5, Task 6B, Task 9B
Mahith Murari - 70994954 - Task 1, Task 2, Task 6A, Task 8
Kartheek Reddy Gade - 58842403 - Task 3A, Task 4, Task 7, Task 9A

All the members have contributed collectively in the brainstorming and design/analysis part, testing with random input files and documentation. Additionally, each person programmed the above tasks with mutual collaboration incase of roadblocks.

Design and Analysis of Algorithms:

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

Problem1
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

Definition:
A is a 2D vector of m*n size representing the predicted prices of m stocks for n days. Consider four variables: stock_index, buy_day, sell_day, max_diff and initialize them to 0,0,0,-1 respectively. Scan the whole input matrix in such a way to find the difference between each and every element. For i=0 till m, for j=0 till n, for k = j+1 till n find the difference i.e., A[i][j]-A[i][k]. At each step if the difference is greater than the max_diff then replace the max_diff's value with this difference and stock_index, buy_day, sell_day to the current pointers respectively. After the complete scan if the max_diff = -1 then return last stocks stock, last day as Buying and Selling day for the max profit else return stock_index, buy_day and sell_day as the stock, Buying day and Selling day for maximum profit.

Proof of Correctness:
By the problem definition we need to find the max profit transaction. Since we can only buy and sell the same stock, our problem reduces to finding out which particular stock yields the maximum profit. Here there is no overlap between stocks.
For i=0 till m stocks, for j=0 till n days, for k = j+1 till n, we find the difference i.e., A[i][j]-A[i][k]. At each step if the difference is greater than the max_diff(maximum profit) then replace max_diff's value with this difference and stock_index, buy_day, sell_day to the current pointers i.e., i,j,k respectively.

Hence we get a particular stock's max_diff and its corresponding sell day is stored. By executing all the problems and updating the max_diff, we get the maximum profit.

Analysis:
In this algorithm, three for loops are used to traverse $i \in [0, m]$ & $j \in [0, n]$ & k $\in [0, n]$, so the time complexity is $O(m*n^2)$. We are using only integers max_diff, i, j and k to store the max value, stockIndex, buy and sell indices of max profit yielding stock. Hence constant additional/ auxiliary space complexity O(1).

Alg2 Design a Θ(m ∗ n) time greedy algorithm for solving Problem1

Definition:
A is a 2D vector of m*n size representing the predicted prices of m stocks for n days. Consider four variables: stockIndex, buyIndex, sellIndex, globalProfit and initialize them to 0,0,0,INT_MIN respectively. Initially, Scan the input matrix row wise i.e., for j=0 till m, and find the lowest stock, highest profit and sell(index where we get the highest profit) from each row and if the globalProfit is less than this highest profit then replace the globalProfit's value with this highest profit and make sellIndex to the sell, stockIndex to j+1 and find the buyIndex by iterating the A matrix and find the index where we get the value A[j][sell]-globalProfit. Finally return the stockIndex, buyIndex+1 and sellIndex+1 as stock, Buying day and Selling day for maximum profit.
Proof of Correctness:
By the problem we need to find the max profit transaction. Since we can only buy and sell the same stock, our problem reduces to finding out which particular stock yields the maximum profit. Here there is no overlap between stocks.
In the Greedy Algorithm, we find the minimum price of the stock i on day j, and compare it with each stock A[i][j] - buy, where i = 0 to m, j = 1 to n, and buy is the lowest value available so far on stock i, and store the global Profit, sellIndex, buyIndex, stockIndex. After executing the profit for every stock, we compare it with global Profit(over m stocks), compare and update the maximum profit into global profit and also the sellIndex, BuyIndex, stockIndex. By finding the minimum buy day and finding the maximum sell day, we will get the maximum profit for the given Stock.

Analysis:
In this algorithm, two for loops are used to traverse $i \in [0, m]$ & $j \in [1, n]$, so the time complexity is $O(m*n)$. We are using only integers max_profit for maximum profit, i for stockIndex and j for buy and sell indices of max profit yielding stock. Hence constant additional space complexity O(1).

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

Memoization(TopDown):

Design:

To solve this problem using the memoization approach we constructed a recursive equation to solve the subproblems. Initially created a 2D array Dp[m][n] to store the max profit of the particular day of a stock. For every particular stock we call recursive to solve max profit for the stock j by using recursive calls with will internal calls for every day i 1 to n.
The maxprofit obtained from the subproblems will be updated in Dp[j][i].

Dp[j][i] = max(f(Dp, A, i,n-1)+ A[i][n] - A[i][n-1], 0), for i = 1 to m. If n>0;
= 0, If n<0;

By backtracking the Dp[j][i], we can get the buy, sell day and the stock by iterating through maxprofit value.

Correctness:
By the problem definition we need to find the max profit transaction. Since we can only buy and sell the same stock, our problem reduces to finding out which particular stock yields the maximum profit. Here there is no overlap between stocks. Our recurrence relation DP[j][i] has 2 choices to choose maximum value from f(Dp, A, i,n-1) + A[j][i] - A[j][i-1] and 0. The second value 0 means the price of stock on the i-th day is higher than all its previous days and hence even though we sell on day i we can never get a profit value > 0. The first one builds upon the previously calculated value of maximum profit till day i-1. So if A[j][i] > A[j][i-1] -> we get a value that is greater than DP[j][i-1] else we would get a smaller value. But finally we are choosing the maximum of all DP[j][i] for all i = [1,n]. Hence we finally store in max_profit only the maximum profit for a particular stock and its corresponding sell day is also stored. By solving all the subproblem which are m*n and updating their max profit we can get the maximum profit. Hence we can say that this algorithm results in the optimal solution.

Analysis:
Here we have m*n subproblems which are not overlapping each other. So the time complexity is $O(m*n)$.
An input matrix with size m*n is used, and the DP matrix created is also of size m*n. Hence the space complexity is $O(m*n)$.

Bottom Up:

Definition:
In this, we pass the 2D vector A, having the number of stocks as rows and number of days as columns to the task1_dp_bottomup method. The globalProfit variable is used to store the profit of a transaction that yields maximum profit across all stocks. And stockIndex, buyIndex and sellIndex are used to store the stock, buy day and sell day respectively of the transaction yielding globalProfit. We initialize a 2 dimensional vector DP of size no of stocks * no of days. This DP array holds the maximum profit values for particular stock j till day i. We make use of the below recurrence relation to store the values in the DP array:

$$DP[j][i] = \begin{cases} \max(D[j][i-1] + A[j][i] - A[j][i-1], 0); & \text{if } i > 1 \end{cases}$$

0; if i = 1

For all 1 <= j <= no of stocks (m) and 1 <= i <= days(n).

The maximum profit value for particular stock j over the n days = max(DP[j][i]) for all 1 <= i <= n(no of days). This max profit for a particular stock is stored in the max_profit variable. And the corresponding sell date is also stored.

And global profit would be compared with this max_profit, and if the former is smaller then we would update globalProfit to max_profit and store the corresponding stock and sell Index. Buy index is calculated by iterating over the array A for the particular stockIndex row. This buy date computation takes O(n) time. Finally after looping over all the stocks, the values of stock, buy date and sell date of the globalProfit transaction are outputted.

Correctness:
The problem statement is to find the one transaction that yields the maximum profit. Since we can only buy and sell the same stock, our problem reduces to finding out which particular stock yields the maximum profit. Here there is no overlap between stocks. Our recurrence relation DP[j][i] has 2 choices to choose maximum value from D[j][i-1] + A[j][i] - A[j][i-1] and 0. The second value 0 means the price of stock on the i-th day is higher than all its previous days and hence even though we sell on day i we can never get a profit value > 0. The first one builds upon the previously calculated value of maximum profit till day i-1. So if A[j][i] > A[j][i-1] -> we get a value that is greater than DP[j][i-1] else we would get a smaller value. But finally we are choosing the maximum of all DP[j][i] for all i = [1,n]. Hence we finally store in max_profit only the maximum profit for a particular stock and its corresponding sell day is also stored. Since we iterate over all days inside the loop for stock iteration, the globalProfit would be compared anupdated after looping all days of each stock.

Analysis:
In this algorithm, two for loops are used to traverse $j \in [1, m]$ & $i \in [1, n]$. Inside the first [1,m] loop we have 1 loop over [1,n] for calculating DP array and another check to compare if max_value is greater than globalProfit. If greater, we will iterate in O(n) worst case time complexity to find the buy date. So the time complexity is $O(m*n)$. An input matrix with size m*n is used, and the DP matrix created is also of size m*n. Hence the space complexity is $O(m*n)$.

Problem2
Given a matrix A of m × n integers (non-negative) representing the predicted prices of m stocks for n days and an integer k (positive), find a sequence of at most k transactions that gives maximum profit. [Hint :- Try to solve for k = 2 first and then expand that solution.]

Alg4 Design a Θ(m ∗ n^2k) time brute force algorithm for solving Problem2

Design:
Algorithm 4 was designed based on the brute force approach where we used recursive calls task2_bruteforce_recursive(A, max, stockIndex, j, k , currStock) to traverse into each possible call.

There are a total of k transactions. For each transaction it requires to check, buy and sell a stock. If we consider we can sell a stock on a particular day ‘j’, we have ‘j-1’ possibilities to buy a stock for ‘k' th transaction and we have the recursion calls which will call for all ‘k’ transactions. If profit from a call is ‘temp’, current profit is currProfit then compare(temp, currProfit) and update currProfit.

Correctness:
Here we traversed through the entire recursive tree of every subproblem. Where each possible day has ‘n’ other days to buy or sell for ‘k’ transactions for ‘m’ stocks. For every recursive call we get profit which is checked with current profit and updated if it is greater than it. This leads to generating a solution with at most k transactions with maximum profit. Therefore we can prove that the algorithm can achieve a solution by considering all possible cases.

Time Complexity:
We have 2 for loops which is n^2, and each possible day for one transaction, for k transactions it is n^2k. For m stocks it O(m*n^(2k)).

Space Complexity:
As we used a 2D vector to backtrack the indices. So, it is O(n^2).

Alg5 Design a Θ(m ∗ n^2 ∗ k) time dynamic programming algorithm for solving Problem2

Design:
To solve this problem, we pass the maximum number of transactions k and 2D vector A, having the number of stocks as rows and number of days as columns as parameters to the task2_dp_bottomup method. We make use of tracker array of size (k+1)*(n+1) size for storing the indices of stock, buy date and sell date that yield maximum profit in string format. The dp array is to store the maximum profit and dp[i][j] represents the max profit for at most i transactions till day j. We use 0 based indexing throughout the problem and add 1 while outputting the values. We make use of the below recurrence relation to store the values in the DP array:

DP[i][j] = 0; if i = 0
= 0; if j = 0
= max(DP[i][j-1], max(A[v][j] - A[v][prevj] + DP[i-1][prevj])); otherwise
and 0 <= prevj < j; 0 <= v < no of stocks.

Correctness:
Base cases are if i=0 means 0 transactions and hence max profit achievable is 0. Similarly if j =0, on the 1st day we can buy and sell yielding a max profit of 0. Hence the values in the tracker array for these indices would be empty because no transactions yield profit > 0. For other values, we have two possibilities. The profit of making ‘i’ transactions till j-1 th day and no sell operation on j-th day. Or to sell on j-th day marking it the i-th transaction and hence we need to find which among the previous j-1 days yields maximum profit if we buy on that day i.e., i-1 transactions are done till some prevj ∈ [0, j-1] and last i-th operation buy day is prevj and sell day is j. So essentially at each step we are checking all the possibilities of sell/not sell at jth day,

and all buy days for the ith transaction if we sell on day i and, if we don't buy then dp[i][j-1] already has the max profit till j-1 for i transactions. So we are building upon the previously calculated optimal solution till previous days and previous transactions. Hence our recurrence relation takes into account all possibilities and computes maximum profits.

Complexity:
The time complexity to compute DP array is O(m*(n^2)*k). We have 3 iterative loops. Inside the topmost loop for k transactions, we loop over all the days for each stock m. While calculating the DP array, we also loop over previous days to find which buying day would yield max profit if we sell on the j-th day. This adds to another O(n) additional time complexity. This could be optimized and is implemented as part of ALG-6.

For the purpose of tracking the indices, if we sell the stock on j-th day to make i-th transaction and buy on prevj day(this prevj day yields max profit), then we update the tracker[i][j] = "stock-index-of-i-th-transaction prevj j" and append tracker[i-1][prev] (which is the track of best transactions till prevj's day). If we dont sell on j-th day, we update tracker[i][j] = tracker[i][j-1]. This way we keep track of the transactions that yield max profit simultaneously while updating the dp array.

Space Complexity:
We are using input array of size m*n and also create tracker array of size (k+1) * (n+1). As per the problem statement the ranges of k and m are similar and hence space complexity is of order O(m * n).

Alg6 Design a $\Theta(m * n * k)$ time dynamic programming algorithm for solving Problem2

(6A) Memoization:

Design:
To solve this problem using memorization we used a recursive call to compute maximum profit with at most k transactions. By passing arguments as 2D 'A' array contains stocks with respective days, transaction number 'k', wheather stock holds or not 'buy', memo to store the key, buy index, previous stock. If memo is filled for a sub problem then it will not allow the same subproblem to be computed in further recursive calls.
If it already holds a stock, it will check form sellday + 1 to n for a particular stock. Check whether the profit is max than the current profit, if yes, update or else keep current profit.
Check for (k-1)th transaction for all the stocks using the recursive calls.
In current stock, ps = task2_dp_topdown(A, i, k - 1, false, memo, stock, -1,-1);
In next stock, pu = task2_dp_topdown(A, i, k - 1, false, memo, stock + 1, -1,-1);
In pervious stock, pd = task2_dp_topdown(A, i, k - 1, false, memo, stock - 1, -1,-1);
If stock is not holded, then we can buy the stock for a day.
For current stock, ps = task2_dp_topdown(A, i + 1, k, true, memo, stock, i,-1);
If able to buy on next stock only on (prevStock != stock +1)
pu = task2_dp_topdown(A, i, k, false, memo, stock + 1, -1,-1)
If able to buy on previous stock only on (prevStock != stock - 1)
pu = task2_dp_topdown(A, i, k, false, memo, stock - 1, -1,-1).

f(n,m,k,buy) = max(f(i,j,k,buy), max(A[stock][i] +
max(f(i,j,k-1,buy=false),f(i,j+1,k-1,buy=false),f(i,j-1,k-1,buy=false)) If buy =true,

= max(f(i,j,k,buy=true)- A[j][i], max(f(i,j,k,buy=true),
f(i,j+1,k,buy=false),f(i,j-1,k,buy=false))) If buy =false,
= 0 If i>=n,
= 0 If k=0,
= 0 If j>=m, j<0,

Correctness:
Consider the base case if i>n means sell or buy day greater than no. of days, k==0 means completion of k transactions, stock>=m, stock<0. For each transaction in k, checking all the subproblems to find the maximum profit. Buy a stock on the 'k'th transaction, 'i' th day, 'j' th stock and check the previous and next stocks and days where the max profit occurs on each transaction{ compare(compare(ps, pu), pd)}. This indicates that we are checking all possible sub problems through the recursive call and finding the optimal solution using memoization.

Complexity:
We are running a recursive equation as the base case parameters are stock number, ith day, 'k' th transaction. It should be 4T(n,m,k) where the time complexity will be O(m*n*k).
We are using an unordered map to store key value pairs and a structure which stores all the subproblems in O(m*n*k) space complexity.

(6B) BottomUp:

Design: To solve this problem, we pass the maximum number of transactions k and 2D vector A, having the number of stocks as rows and number of days as columns as parameters to the task2_dp_bottomup method. We make use of tracker array of size (k+1)*(n+1) size for storing the indices of stock, buy date and sell date that yield maximum profit in string format. The dp array is to store the maximum profit and dp[i][j] represents the max profit for at most i transactions till day j. We use 0 based indexing throughout the problem and add 1 while outputting the values. We make use of the below recurrence relation to store the values in the DP array:

DP[i][j] = 0; if i = 0
= 0; if j = 0
= max(DP[i][j-1], A[v][j] - A[v][j-1] + DP[i-1][j-1])); otherwise, where 0 <= v < no of stocks

Correctness:
Base cases are if i=0 means 0 transactions and hence max profit achievable is 0. Similarly if j =0, on the 1st day we can buy and sell yielding a max profit of 0. For other values, we have two possibilities. The profit of making 'i' transactions till j-1 th day and no sell operation on j-th day. Or to sell on j-th day marking it the i-th transaction and hence we need to find which among the previous j-1 days yields maximum profit if we buy on that day i.e., i-1 transactions are done till some day prevj $\in$ [0, j-1] and last i-th operation buy day is prevj and sell day is j. So essentially

at each step we are checking all the possibilities of sell/not sell at jth day, and all buy days for the ith transaction if we sell on day i and, if we don't buy then dp[i][j-1] already has the max profit till j-1 for i transactions. So we are building upon the previously calculated optimal solution till previous days and previous transactions. Hence our recurrence relation takes into account all possibilities and computes maximum profits.

Complexity:
The time complexity to compute DP array is O(m*n*k). We have 3 iterative loops. Inside the topmost loop for k transactions, we loop over all the days for each stock m. This is an optimization of the algorithm implemented as part of ALG-5.To reduce the additional O(n) complexity for identifying which buying day would yield max profit if we sell on the i-th day, we alter the DP recurrence relation. Inside the loop for days, let j' be the buy day for sell on j-th day for i-th transaction. For each stock v, we check if - A[v][j-1] + dp[i - 1][j-1] is greater than the previous day's value of - A[v][j'-1] + dp[i - 1][j'-1] and if it is greater, we update the buy day to j from j'.

For the purpose of tracking the indices, if we sell the stock on j-th day to make i-th transaction and buy on some prevj day(this prevj day yields max profit), then we update the tracker[i][j] = "stock-index-of-i-th-transaction prevj j" and append tracker[i-1][j-1] (which is the track of best transactions till prevj's day). If we don't sell on j-th day, we update tracker[i][j] = tracker[i][j-1]. This way we keep track of the transactions that yield max profit simultaneously while updating the dp array.

We are using input array of size m*n and also create DP, stockIndex, buyIndex and sellIndex arrays of size (k+1) * (n+1). As per the problem statement the ranges of k and m are similar and hence space complexity is of order O(k * n).

BONUS
Problem3

Given a matrix A of m × n integers (non-negative) representing the predicted prices of m stocks for n days and an integer c (positive), find the maximum profit with no restriction on the number of transactions. However, you cannot buy any stock for c days after selling any stock. If you sell a stock at day i, you are not allowed to buy any stock until day i+c+1

ALG7
Design a Θ(m ∗ 2^n) time brute force algorithm for solving Problem3

Design:
To solve this problem using brute force method we run every possible combination to get maximum possible profit out of 'm' stocks and 'n' days. For every day we have two choices: either we buy or sell on that day. If we buy a stock on 'i'th day then we have 'i+1' to 'n' possible ways. If we sell a stock on 'i'th day we can only buy the next stock after 'i+c+1' days.

We have recursive calls to run them* 2^n subproblems for a particular stock. They are:

Profit = max(profit, f(false, 0, A[0].size(), A[i], c, optimal, i)); for m stocks we have to compute it for each day considering that we have the possibility to buy or sell stock on that day.

f(brought, i, n, A, c, optimal, stock) = max(f(brought, i+1, n, A, c, optimal, stock), A[i]+f(false, i + c + 1, n, A, c, optimal, stock)) If brought = true,

= max(f(brought, i+1, n, A, c, optimal, stock), f(true, i + 1, n, A, c, optimal, stock)-A[i])) If brought = false.

Correctness:

Consider the algorithm that generates the solution which is not optimal and there exit a solution. Now we try to prove this by contradiction. Here for everyday I can have two possible options: day and sell, there are 'm' stocks we have to run for it too. Running the recursive equation everyday for every stock has buy and sell Profit = max(profit, f(false, 0, A[0].size(), A[i], c, optimal, i). It will run for the subproblems to get the max profit as were the cases stock already holds then we have a chance to sell on this day or subsequent days then again we buy only on 'i+c+1'th day. If it does not hold, I will buy for that day or subsequent days and compute the max profit. Hence we conclude that we achieve max profit from your solution. So, we can say that our solution produces the optimal solution by contradiction.

Complexity:

We are running a recursive equation as the base case parameters are stock number, n day and m stocks . Everyday has 2 possibilities buy and sell, so n days can have 2^n problems and for m days it will be m*2^n. Where the time complexity will be O(m*2^n).

We are using an unordered map to store key value pairs and a structure which stores all the subproblems in O(m*n) space complexity.

ALG 8

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

Design:

To solve this problem using the tabulation method we are using a 2D vector of string type, in this we are storing the maximum possible profit for that day from all the possible cases. Initially we are taking store[][] to store transaction as null, maxi[][] we have maximum values as zeros, after each iteration for stocks 1 to m we are computing maxprofit for each day('i' i.e 1 to n) as a sell day and considering 'i-1' days where we have possibility to buy a stock and computing the max profit by using the maximum values of the previous transactions. We backtracking the possible solution by using maxi[][] values which will refer to transaction details in store[][].

maxi[i][j] = maxi(A[s][i]-A[s][j], maxi[i][j]), if maxi[i][j] is less than the any set of transaction till buy day 'i' and sell days till to j, we will update maxi[i][j] with that profit and we also update store[i][j] = to_string(s+1) + " " + to_string(j+1) + " " + to_string(i+1).

Now, we have to check for the optimal solution from maxi[i][0] which means to sell at 'i' and consider buying at i-1 to 0 plus maxi[prevIndex][0] or existing maxi[i][0].

maxi[i][0] = max(maxi[i][j] + (prevIndex < 0 ? 0 : maxi[prevIndex][0]), maxi[i][0]) and storing respective buy and sell index in

store[i][0] = store[i][j] + (prevIndex < 0 ? "" : (store[prevIndex][0] != "" ? "\n" + store[prevIndex][0] : ""))

maxi[prevIndex][0]) in this 'prevIndex'means sellIndex-(c+1)

Correctness:
We consider the base condition as 0<=stock<=m, 0<=day<=n, and we are storing maximum profit for every day. We are storing each and every transaction when we update the max[i][j] and store store[i][j]. We are computing the maxi[i][j] by considering the all subproblem maxi[i][i-1] to maxi[i][j]. By following this approach we will get max profit of 'i' which will be updated in maxi[i][0] and considering this we can update buy and sell indices in store[i][0]. Now computing maxi[i][j] = maxi(A[s][i]-A[s][j], maxi[i][j]), which can achieve maximum profit for 'i'th day by doing this for n days and m stocks. Hence we can prove that by using this bottomUp approach we can get an optimal solution.

Complexity:
We are computing 'i' th day profit by filling all the sub problems till 'i-1' for 'm' stocks in 2D vector. We have run loop till n and inner loop n-1 and for 'm' stocks. So, time complexity is O(m*n^2).
We are using an additional space to compute the maximum profits of n days considering n days as buy days. So, space complexity is O(n^2).

ALG 9
Design a Θ(m ∗ n) time dynamic programming algorithm for solving Problem3

(9A) Memoization:

Design:
To solve this problem using memorization we used a recursive call to compute maximum profit out of 'm' stocks in 'n' days with a cooldown period as 'c'. By passing arguments as 2D 'A' array contains stocks with respective days, cooldown period 'c', wheather stock holds or not 'buy', memo to store the key, buy index, previous stock. If it computed the value then we use it or else we get the value from subproblems.
if(memo.find(key) == memo.end()) this will avoid recomputing the solved subproblems.
If it already holds a stock, it will check form sellday + c + 1 to n for a particular stock. Because we are only able to buy new stock after the cooldown period (i+c+1). Check whether the profit is max than the current profit, if yes, update or else keep current profit.
Check for all the stocks using the recursive calls to solve m*n subproblems.
In current stock, ps = task3_dp_topdown(A, i+c+1, c, false, memo, stock, -1,-1);
In next stock, pu = task3_dp_topdown(A, i+c+1, c, false, memo, stock + 1, -1,-1);
In pervious stock, pd = task3_dp_topdown(A, i+c+1, c, false, memo, stock - 1, -1,-1);
If stock is not holded, then we can buy the stock for a day.
For current stock, ps = task3_dp_topdown(A, i + 1, c, true, memo, stock, i,-1);
If able to buy on next stock only on (prevStock != stock +1)
pu = task3_dp_topdown(A, i, c, false, memo, stock + 1, -1,-1)
If able to buy on previous stock only on (prevStock != stock - 1)
pu = task3_dp_topdown(A, i, c, false, memo, stock - 1, -1,-1).

f(n,m,c,buy) = max(f(i+1,j,c,buy), max(A[stock][i] +
max(f(i,j,i+c+1,buy=false),f(i,j+1i+c+1,buy=false),f(i,j-1,i+c+1,buy=false)).

If buy=true,

= max(f(i+1,j,c,buy=true)- A[j][i], max(f(i,j,c,buy=true),
f(i,j+1,c,buy=false),f(i,j-1,c,buy=false))) If buy =false,
= 0 If i>=n,
= 0 If j>=m, j<0,

Correctness:
Consider the base case if i>n means sell or buy day greater than no. of days, stock>=m, stock<0. For each subproblem, check all the possible cases to find the maximum profit. We can compute the values for subproblems and store them to solve bigger subproblems. Buy a stock on the 'i' th day, 'j' th stock and check the previous and next stocks and days where the max profit occurs on each transaction{ compare(compare(ps, pu), pd)} after sell we have to cool it down till next 'c+1' days to buy new stock. By using recursion we can solve the sub problems of 'm*n'. From which we can get the max profit after every recursive call whether current profit is greater than the obtained. This indicates that we are checking all possible sub problems through the recursive call and finding the optimal solution using memoization.

Complexity:
We are running a recursive equation as the base case parameters are stock number, n day and m stocks . It should be 4T(n,m,) where the time complexity will be O(m*n).
We are using an unordered map to store key value pairs and a structure which stores all the subproblems in O(m*n) space complexity.

(9B) Bottom Up:

Design:
To solve this problem by using BottomUp approach we created an unordered map to store the key value pair 'dp', where we store the values in dp of sell transaction and buy transaction. If we hold stock we will compute the dp[sk] buy comparing all subproblems and getting the max profit. Make the stock hold for next day, skip = (dp.find(to_string(v) + " " + to_string(i+1) + " " + to_string(true)) != dp.end()) ?dp[to_string(v) + " " + to_string(i+1) + " " + to_string(true)]: a; then compute the max profit for 'i'. If we don't hold the stock then make buy = true and compute for remaining stocks of the same day skip = (dp.find(to_string(v) + " " + to_string(i+1) + " " + to_string(false)) != dp.end())?dp[to_string(v) + " " + to_string(i+1) + " " + to_string(false)]: a; and compute the remaining subproblems.

dp[v, n-1,buy] = max(dp[v, i+1, true],
A[v][i]+max(dp[v+1,i+c+1,flase],dp[v,i+c+1,false],dp[v-1,i+c+1,false]))
If buy = true,
= max(dp[v, i+1, false],
max(dp[v+1,i+c+1,flase],A[v][i]+dp[v,i,true],dp[v-1,i,false]))
If buy = false,
= 0 If i<0,v<0

Correctness:

The base condition is i<0, v<0 means no day, no stock respectively.Then we have come with a subproblem where, If we hold stock we will compute the dp[sk] buy comparing all subproblems and getting the max profit. Make the stock hold for next day, skip = (dp.find(to_string(v) + " " + to_string(i+1) + " " + to_string(true)) != dp.end()) ?dp[to_string(v) + " " + to_string(i+1) + " " + to_string(true)]: a; then compute the max profit for 'i'. If we don't hold the stock then make buy = true and compute for remaining stocks of the same day skip = (dp.find(to_string(v) + " " + to_string(i+1) + " " + to_string(false)) != dp.end())?dp[to_string(v) + " " + to_string(i+1) + " " + to_string(false)]: a; and compute the remaining subproblems. This can compute the subproblems and stores the values of m*n subproblems. This indicates that we are solving all the subproblems and getting the optimal solution.

Complexity:
We are computing the subproblems of m*n. Where the time complexity will be O(m*n).
We are using an unordered map to store key value pairs and a structure which stores all the subproblems in O(m*n).

Experimental Comparative Study:

PLOT -1

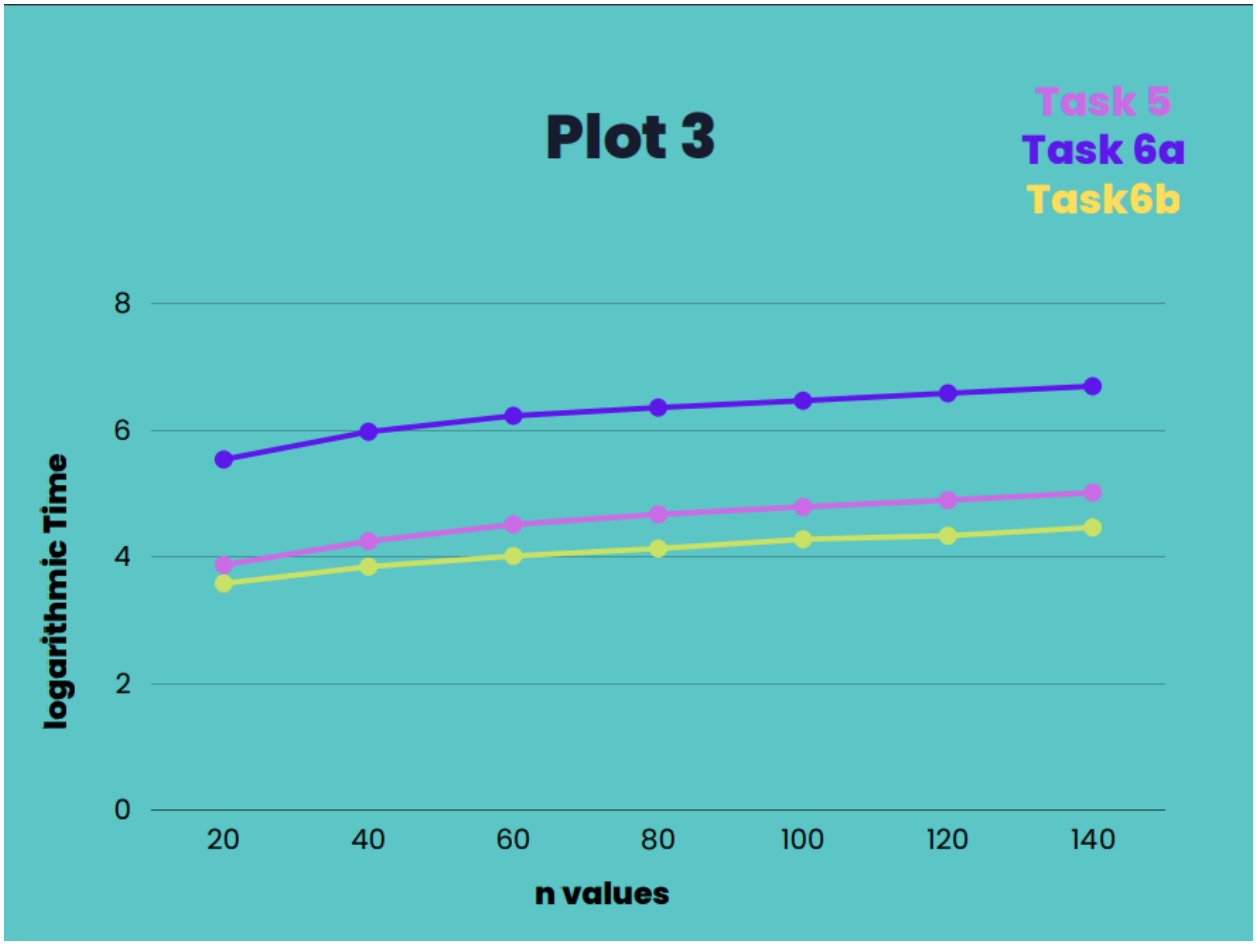
Y-axis -> log to base 10 (runtime in microseconds)
X axis -> n values
Fixed m = 50 and variable n

PLOT -2
Y-axis -> log to base 10 (runtime in microseconds)
X axis -> m values

Fixed n = 5000 and variable m

PLOT-3
Y-axis -> log to base 10 (runtime in microseconds)
X axis -> n values

Fixed k = 15, m =25 and variable n



PLOT - 4
Y-axis -> log to base 10 (runtime in microseconds)
X axis -> m values

Fixed k = 15, n = 500 and variable m

PLOT-5
Y-axis -> log to base 10 (runtime in microseconds)
X axis -> k

Fixed n = 25, m = 100 and variable k

PLOT-6

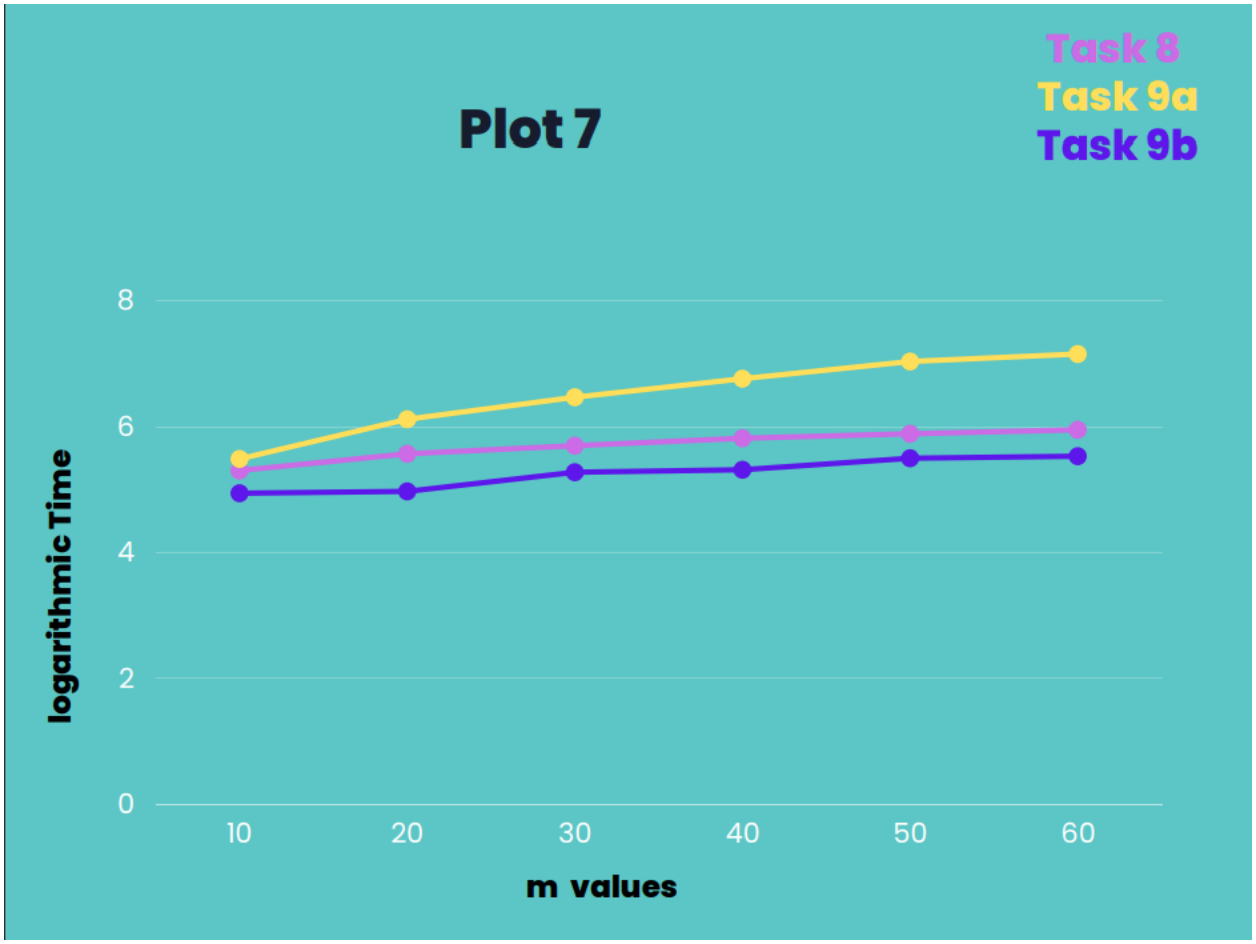
Y-axis -> log to base 10 (runtime in microseconds)
X axis -> n values

Fixed c = 5 , m = 25 and variable n

PLOT - 7

Y-axis -> log to base 10 (runtime in microseconds)
X axis -> m values

Fixed c = 5 and n = 500 and variable m



Graph Trends and Observations/ Conclusions:

- For problems 2 and 3, we were unable to run the brute force algorithm as it takes exponential time and could not compute outputs for the inputs used to draw the graphs. But we were able to verify the outputs of the brute force algorithm for smaller k,n and m values. Because for problem 2 its time complexity is in the exponential factor which is 2 powers. We failed at input (k=5, m=10, n=10) and hence lack enough values to compare with remaining trends.

- We have plotted the graphs using log to base 10 (values of run time in microseconds) versus the variable parameter for all the plots.

- In Plots 1 and Plots 2,the task 2 Greedy algorithm performs better than the brute force and DP algorithms. The DP algorithms compute and keep track of all the subproblems irrespective of if they yield maximum profit. But the greedy algorithm keeps track of only the transactions that yield maximum profits/ values, hence saving additional space and time complexity allowing it to run faster than dynamic programming algorithms.

- We faced difficulty in backtracking in the top down approach for problem 6a and also in 9a, to find the stocks and corresponding buy and sell indices that yield the maximum profit. Hence we used TradeInfo struct which has fields to store the sell day, profit value and the series of transactions that yield the profit value stored in struct till the sellDay i.e., for all sub problems we are storing the profits and the indices values that yield max profit till that sellIndex.

- Also since we calculate and store in the DP array each and every possible traversal path i.e., “sell-start-stock” values, this additional overhead of traversing each key is causing the top down approaches in problem 2 i.e., task 6a to run slower than the O(m* n^2 * k) task 5 as observed in plots 3, 4 and 5. If we just compute the top down approach without storing the indices of buy and sell indices for max profit yielding stocks, it runs faster than task 5.

- We observed a challenge designing and running the brute force for problem 4a, where the algorithm is broken down for larger k values due to the complexity. It was challenging to come up with a brute force algorithm that runs in the O(m * n^ (2k)) for algorithm 4. We had to iterate, observe and convert the O(m * 2^n) to run in the required time complexity