1. Team Members

Naganaboina Venkata Sai Dheeraj (UFID: 35186979): Worked on Task1, Task3B, Task6B, Task4, experimental comparative study and testing.
Suraj Ramadugu (UFID: 80155074): Worked on Task5, Task 6A, Task 4, testing, Makefile and experimental comparative study.
Pravallika Maddukuri (UFID: 59807014): Worked on Task2, Task3A, , Task4, project report and experimental comparative study.

2. Design and Analysis of Algorithms

Task 1: $\Theta(m*n^2)$ time brute force algorithm for solving problem 1

By using 3 nested loops, we tried to find the maximum possible profit from all the stocks by making one transaction. The outermost loop 'i' is used to iterate over 'm' stocks and the rest two inner loops 'j' and 'k' are used to find the maximum profit each of those 'm' stocks for 'n' days. We used a temporary variable 'maxProfit' for storing the maximum profit obtained till the current iteration, which gets updated every time when there is a profit greater than 'maxProfit'. We kept track of the stock index 'stock_idx', buy date 'buyDay_idx' and sell date 'sellDay_idx' indices for maximum profit.

Algorithm:

```
Initialize stock_idx, buyDay_idx, sellDay_idx to -1
Initialize maxProfit to 0
For iterations i=0 to number of stocks
        For iterations j=0 to number of days
                For iterations k=j+1 to number of days
                        Set current profit as (prices[i][k] - prices[i][j])
                        if current profit is greater than maxProfit
                                set maxProfit to profit
                                set stock_idx to i
                                set buyDay_idx to j
                                set sellDay_idx to k
Print stock_idx+1, buyDay+1, sellDay_idx+1
```

Proof of Correctness:

First, we initialize stock_idx, butDay_idx, sellDay_idx to -1 and profit which is a temporary variable to store local maximum to 0 and maxProfit which stores global

maximum to 0. The two inner loops find the maximum profit obtained from 1 stock by checking all possible combinations and stores it in the profit variable along with the buy index and sell index and the outer loop iterates over all the stocks and updates the global maximum i.e. the maxProfit variable and also updates the sell index and but index. So, by the end of three loops, we get the maximum profit, buy index and sell index in the respective variables.

Time Complexity:

As there are 3 loops one for number of stocks and the remaining for number of days, so the time complexity is:
Time complexity for iterating over number of stocks =O(m)
Time complexity for iterating two loops over number of days=O(n*n)
Therefore, the time complexity is O(m*n^2)

Space Complexity:

The maximum input we are giving to the algorithm is the prices array which is having ‘m’ rows and ‘n’ columns, so the space complexity is O(1).

Task 2: Θ(m*n) time greedy algorithm for solving problem 1

In this approach we are greedy for profit. We try to find the maximum profit, in the process of increasing the profit, the purchase rate should be less and the sell rate should be maximum. So we initialize buy cost 'buyPrice' to the first value of the prices array. Then while iterating in price array 'prices[][]' we will be checking if the current buy cost 'buyprice' is minimum then the purchase should be made on the current date, else in the case of the current price 'buyPrice' is more than the predecessor cost 'prices[i][j]', then calculate the profit and set that value to maximum profit 'maxProfit'.

Algorithm:

```
Initialize stock_idx, buyDay_idx, buy day, buyDay_idx_var, sellDay_idx to -1
Initialize maxProfit to 0
For iterations i=0 to number of stocks
        Initialize buyPrice to first element of the prices array
        Initialize buy day dummy index to 0
        For iterations j=1 to number of days
                if (buyPrice is greater than prices[i][j])
                        set buyPrice to prices[i][j]
                        set buy day dummy index to j
                else if(prices[i][j]-buyPrice greater than maxProfit)
                        set maxProfit to prices[i][j]-buyPrice
                        set stock_idx to i
                        buyDay_idx to buyDay_idx_var
                        sellDay_idx to j
Print stock_idx+1, buyDay+1, sellDay_idx+1
```

Proof of Correctness:

We initialize stock_idx, buyDay_idx, buyDay_idx_var, sellDay_idx to -1, maxProfit to 0. In every iteration, buyPrice is initialized to the price of a stock on the first day and buy day dummy index to the first day. In the next loop, if the price on next day is lower than the current buy price, the buy price is updated to the lower value along with the buy day dummy index. If the price on one of the next days is greater than the buy price, then we calculate the profit and store it in the maxProfit variable as it is the maximum profit

obtained on one stock till now and we also update the buy index and sell indeed. The outer loop runs this logic on all the stocks and by the end of it, maxProfit variable will hold the maximum profit obtained over all the stocks and buyDay_idx will hold the value of index if date of purchase and sellDay_idx will store the index of date of sale.

Time Complexity:
Time complexity for iterating over number of stocks=O(m)
Time complexity for iterating over number of days=O(n)
As the maximum time complexity after the if and else if loop=O(m*n)
So the total time complexity is O(m*n)

Space Complexity:
The maximum input we are giving to the algorithm is the prices array which is having ‘m’ rows and ‘n’ columns, so the space complexity is O(1).

Task 3A: Θ(m*n) time recursive implementation for solving problem 1 using Memoization.

For this problem we use top down approach where optimal sub structure values are store in dp_max_profit array that holds maximum profit obtained on a given day. This array is first initialized to -1 and then filled from nth day and we recursively calculate profit for n-1 th day and store the results in dp_max_profit. When dp_max_profit is filled we iterate over the array to find the indices of stock, buy date and sell date indices. The recurrence relation to solve this problem is:

max(max_profit(dp_max_profit,prices,m,n-1) + prices[m][n] - prices[m][n-1], 0)

We buy at dp_max_profit element is 0 and try finding the maximum profit obtained in each row. Maximum of obtained from all rows contains the transaction that we need. Hence we print the indices of that transaction.

Algorithm:

Initialize dp_max_profit[i][j] to -1
call recursive function max_profit(dp_max_profit, prices, i, days-1)

```
public int max_profit(int[][] dp_max_profit, int[][] prices, int m, int n) {
    if(dp_max_profit[m][n] == -1)
            dp_max_profit[m][n] =
        Math.max(max_profit(dp_max_profit,prices,m,n-1) + prices[m][n] -
        prices[m][n-1], 0);
    return dp_max_profit[m][n];
```

Proof of Correctness:

In this recursive implementation, we initialize a temporary matrix to store the subproblems. The max_profit method fills one row at a time in the temporary matrix in a top down manner i.e. from n to 1. This method is called in a loop to run over all the stocks. So, every element i in the temporary matrix is equal the maximum of 0 and sum of max_profit till i-1 and price of the stock on ith day and -(price of stock on (i-1)th day).

Once we find the stock that produces maximum profit we can track the buy date and sell date indices using the temporary variable.

Time Complexity:
The time complexity to run this algorithm is in the order of number of stocks * number of days. So, the overall time complexity is O(m*n).

Space Complexity:
A temporary array of size number of days(m*n) called dp_max_profit is used and some other temporary variables are created to hold temporary values. So, the space complexity is O(m*n).

Task 3B: Θ(m*n) time dynamic programming implementation for solving problem 1 using Bottom - Up approach.

For this problem, we use a bottom up approach where a 'ans' array is initialized that holds the maximum profit obtained on a given day. This array is filled to get the maximum profit in its last index and the best buy date and sell date are backtracked using this array. The prices of the stocks are stored in the 'prices' matrix. The recurrence relation to solve this problem is:

ans[i] = max{prices[k][i] - prices[k][i-1] + ans[i-1], prices[k][i] - prices[k][i-1]}

We use a 'temp' variable to store the temporary maximum profit and a variable 'temp_sell_inx' to store the temporary best sell date index. After the whole ans array is filled, we are left with maximum profit, the stock index on which the best transaction is being performed and sell date and we backtrack to find the best buy date to maximize the profit.

Algorithm:

```
maxProfit:
Initialize temp to 0, temp_sell_idx to -1, maxProfitValue = 0
Iterate k over the range (0, number of stocks):
        Initialize ans[]
        Iterate i over the range (1, number of days):
                ans[i] = max{prices[k][i] - prices[k][i-1] +ans[i-1], prices[k][i] -
prices[k][i-1]}
                if temp < ans[i]:
                        temp = ans[i]
                        temp_sell_idx = i
        if temp > maxProfitValue:
                maxProfitValue = temp
                stock_idx = k
                sellDay_idx = temp_sell_idx
```

Proof of Correctness:

First, we initialize temp which is a variable to hold local maximum profit to 0, temp_sell_idx to -1 and maxProfitVal to 0. In the outermost loop which goes from 0 to number of stocks, initialize a temporary array called as ans that stores the maximum profit values until a given day. In the inner loop that iterates over 1 to number of days, we update values in ans array using the recurrence relation that finds whether it is better to complete a transaction on a given day or to skip that day. Every updated value of ans

array is compared to the temporary profit and if it greater than the temporary profit, then the temp variable and temp_sell_idx variable are updated. After the execution of loops, we check if temp is greater than maximum profit and if it is greater, we update the maxProfitVal to temp and also assign values to stock_idx and sellDay_idx. Using this information we can backtrack to get the best buy date in O(n) time.

Time Complexity:

The time complexity to run this algorithm is in the order of number of stocks * number of days. and the time complexity to backtrack the solution is in the order of number of days. So, the overall time complexity is O(m*n).

Space Complexity:
A temporary array of size number of days(=n), called ans is used and some other temporary variables are created to hold temporary values. So, the space complexity is O(n).

Task 4: Θ(m*n^(2*k)) time recursive implementation for solving problem 2.

Here we use a brute force approach to solve this problem. In this approach we do not store any results and calculate every subproblem and find the result. In this we mainly used 2 cases which recursion case 1 to check if the share is not bought and case 2 is if we bought stock and then with recursion we find max profit with each subproblem. And find the selling stock with maximum profit. We repeat this process for maximum k transactions. Recurrence relation is

if stock is bought max(res,maxstock(A,i+1,u,remaining)-A[u][i]) while u iterates from o to m
if stock is not bought (either included in profit or the stock is left) hence we use
max(res,maxstock(A,i+1,u,remaining-1)-A[u][i]+A[bought][i])
max(res,maxstock(A,i+1,-1,remaining-1)+A[bought][i]) to find maximum result while u iterates from o to m

Algorithm:

```
public int maxstock(int A[][],int i,int bought,int remaining)
          base case to check (i==n) || remaining==0
            if bought  is -1 && remaining is 0  return 0
          else return min_value
    int res=min_value;
 if(bought==-1)
  iterate u from 0 to m
  res=Math.max(res,maxstock(A,i+1,u,remaining)-A[u][i]);
else
   iterate u from 0 to m
      if(u!=bought)
        res=max(res,maxstock(A,i+1,u,remaining-1)-A[u][i]+A[bought][i])
       res=max(res,maxstock(A,i+1,-1,remaining-1)+A[bought][i]);
  res=max(res,maxstock(A,i+1,bought,remaining))
return res
```

Proof of Correctness:

Base case is if profit is computed till n days or if maximum allowed transactions are completed, then return.The variable bought gives us the information whether a stock is purchased or not. If a stock is not purchased on a given day, the resulting profit would be the maximum profit obtained for a transaction that might have ended on the previous day. If a stock is purchased, the profit would be the sum of maximum profit till now - price of that stock on that day. If a stock is sold, the profit would be equal to the sum of maximum profit till now + price of stock on sell date - price of stock on buy date. Also, the number of allowed transactions is reduced by one in this case. So, finally the res variable will store the maximum profit.

Time Complexity:

Time complexity to run the whole algorithm through recurrence relation is O(m*n^(2*k)). As for each k value we iterate n^k times;
So the overall time complexity is O(m*n^(2*k)).

Space Complexity:

We don’t use any extra memory so the space complexity of this solution is O(1).

Task 5: Θ(m*n^2*k) time dynamic programming algorithm for solving problem 2.

We use the bottom up approach to solve this problem. We create a temporary table 'T', that stores the maximum profit at every iteration based on the maximum number of transactions being allowed. If the price per day of stocks is denoted in the 'price' table, the recurrence relation is:

T[i][j] = max{T[i][j-1], max(price[k][j] - price[k][m] + T[i-1][m]}
where m goes from 0 to j-1 and k does from 0 to number of stocks.

For every transaction, we consider all the possible transactions that can be performed until that day to find the best transaction. After the whole table T is filled, the last element gives the maximum profit and we backtrack to print the solution i.e., which stock is bought on which day and sold on which. The first row and the first column of table T are 0 as you cannot perform a transaction on 1st day as you don't have a stock to sell and you cannot complete a transaction if you are not allowed to perform a transaction.

Algorithm:

```
maxProfit:
Iterate i over the range (1, number of stocks):
        Iterate j over the range (1, number of days):
                Initialize maxVal = 0
                Iterate l over the range (0, number of stocks):
                        Iterate m in the range (0, j):
                                maxVal = max{maxVal, prices[l][j] - prices[l][m] +
                        T[i-1][m]}
                T[i][j] = max{T[i][j-1], maxVal}
```

Proof of Correctness:

At every iteration j,we consider the sub problem of all the stocks until jth day. The two possible scenarios on a given day is either we can complete a transaction on that day or we can skip that day. This is the same approach taken in the recurrence relation to find the maximum profit. If we chose to complete a transaction on a given day, we need to calculate the maximum profit that can be obtained by iterating over all the possible buy dates that are less than sell date and once a buy date is decided we also need to add the maximum profit that is obtained by ending a transaction on the decided buy date. This is calculated in the maxVal variable. So, we fill a temporary table by following this approach to get the maximum profit on a given day. As these are the only possible scenarios, by the time we complete filling the temporary table, we are left with the maximum profit possible in the last column of the table.

Time Complexity:

Time complexity to run the whole algorithm and fill the temporary table is number of stocks * (number of days)^2 * number of transactions which is equal to O(m*n^2*k) and the complexity to backtrack the solution is O(m*n^2).
So the overall time complexity is O(m*n^2*k).

Space Complexity:

A temporary table is used to keep track of the maximum profit for any given day and a specified number of transactions which is of thee size number of stocks * number of days. So, the space complexity is O(m*n).

Task 6A: Θ(m*n*k) time dynamic programming algorithm for solving problem 2 using Memoization.

To solve this problem we optimize the solution presented for Task 4 and reduce the time complexity by not checking all the possible sub structures of the recurrence relation to find the best buy date for a transaction. This is a top- down approach and the recurrence relation is:

from 1 to m stocks iterating with recursion
max(calculateProfit(m, n, k-1, prices, maxDiff, dp) - prices[i][n],maxDiff[k][i]) and maximum is max(prices[i][n] + maxDiff[k][i],maximum)

To fill dp array we recursively calculate maxDiff and maximum profit at that recurrence relation using the calculateProfit function. As we memoize the data in dp array if same recurrence relation occurs then instead of calculating the whole substructure again we use the dp array to determine the result hence we reduce the complexity from m*n^(2*k) to m*n*k.

Algorithm:

initialize dp array and fill in -1 except 1st row and column which are initialized to 0
initialize maxDiff array to -prices[i][0]
calculateProfit(m, n-1, k, prices, maxDiff, dp)

```
static int  calculateProfit(int m, int n, int k, int[][] prices, int[][] maxDiff, int[][] dp
)
        if dp[k][n] is equal to -1
                maximum = calculateProfit(m, n-1, k,prices, maxDiff, dp)
                iterate over i to m stocks
                        maxDiff[k][i] = Math.max(calculateProfit(m, n, k-1, prices,
                        maxDiff, dp) - prices[i][n],maxDiff[k][i] )
                        maximum = Math.max( prices[i][n] +
        maxDiff[k][i],maximum);
         dp[k][n] = maximum;
         return maximum;
    return dp[k][n];
```

Proof of Correctness:

First, initialize dp array to -1 except for 1st row and 1st column which are initialized to 1. Next initialize the maxDiff matrix to -(prices[i][0]). The calculateProfit method fills the dp array in a top down fashion. For every ith entry in dp matrix, the maximum profit obtained till (i-1)th day over all the stocks is calculated and to optimize the computation process, maxDiff matrix is used instead of calculating all the previous transactions every

time. This approach would assure to have the maximum profit in the dp array and we can easily backtrack the solution.

Time Complexity:

The time taken to fill the table T is m*n*k as from the recurrence relation. So, the overall time complexity is equal to O(m*n*k).

Space Complexity:

A temporary table is used to keep track off maximum profit obtained at any point which is of size m*n and maxDiff size used is m*n. So, the overall space complexity is O(m*n).

Task 6B: Θ(m*n*k) time dynamic programming algorithm for solving problem 2 using bottom - up approach.

To solve this problem we optimize the solution presented for Task 5 and reduce the time complexity by not checking all the possible buy dates to find the best buy date for a transaction. This is a bottom - up approach and the recurrence relation is:

T[i][j] = max{T[i][j-1], max(price[k][j] + maxD\iff)}, where k goes from 0 to number of stocks and maxDiff = max{maxDiff, T[i-1][j] - prices[k][j]}.

To fill the table T, for every column, we make use of the maximum profit obtained till that day plus the best we can do by ending a transaction on the buy date which is present in the row above the one we are filling. We make use of an array called 'maxDiff' which holds the maximum difference attained before the transaction that can be used to find the maximum profit. Values in the maxDiff array are updated at every iteration to reflect the new best maximum differences that are used to fill the next value in matrix T. m = number of stocks, n = number of days, k = number of transactions allowed.

Algorithm:

```
maxProfit:
Iterate i over the range (1, number of stocks):
        Initialize maxDiff[]
        Iterate l over the range (0, number of stocks):
                Initialize maxDiff[l] = -prices[i][0]
        Iterate j over the range (1, number of days):
                T[i][j] = T[i][j-1]
                Iterate k over the range (0, number of stocks):
                        T[i][j] = max{T[i][j], prices[k][j] + maxDiff[k]}
                        maxDiff[k] = max{maxDiff[k], T[i-1][j] - prices[k][j]
```

Correctness:

In the outermost loop that goes from 1 to the number of stocks. we initialize maxDiff which is a 1 dimensional array that will hold the maximum difference for each of the stocks on a given day. This is useful to compute the profit that will be obtained if a transaction is ended on a given day by just adding the price on that day to the corresponding maxDiff value of that stock. The maxDiff values are initialized to -(value of the stock on day 1) and updated after every day to calculate the profit for the next day. The profit obtained here is compared with the profit obtained till the preceding day as the possible scenarios here are to either complete a transaction on that day or skip that day. So, at every iteration we get the maximum possible profit that can be obtained till that

day over all the stocks. This would guarantee that the last column of the temporary table we are filling will have the maximum profits.

Time Complexity:

The time taken to fill the table T is O(number of stocks * number of days * number of transactions). So, the overall time complexity is equal to O(m*n*k).

Space Complexity:

A temporary table is used to keep track off maximum profit obtained at any point which is of size m*n, an array to hold and update maximum difference which is of the size m. So, the overall space complexity is O(m*n).

3. Experimental Comparative Study

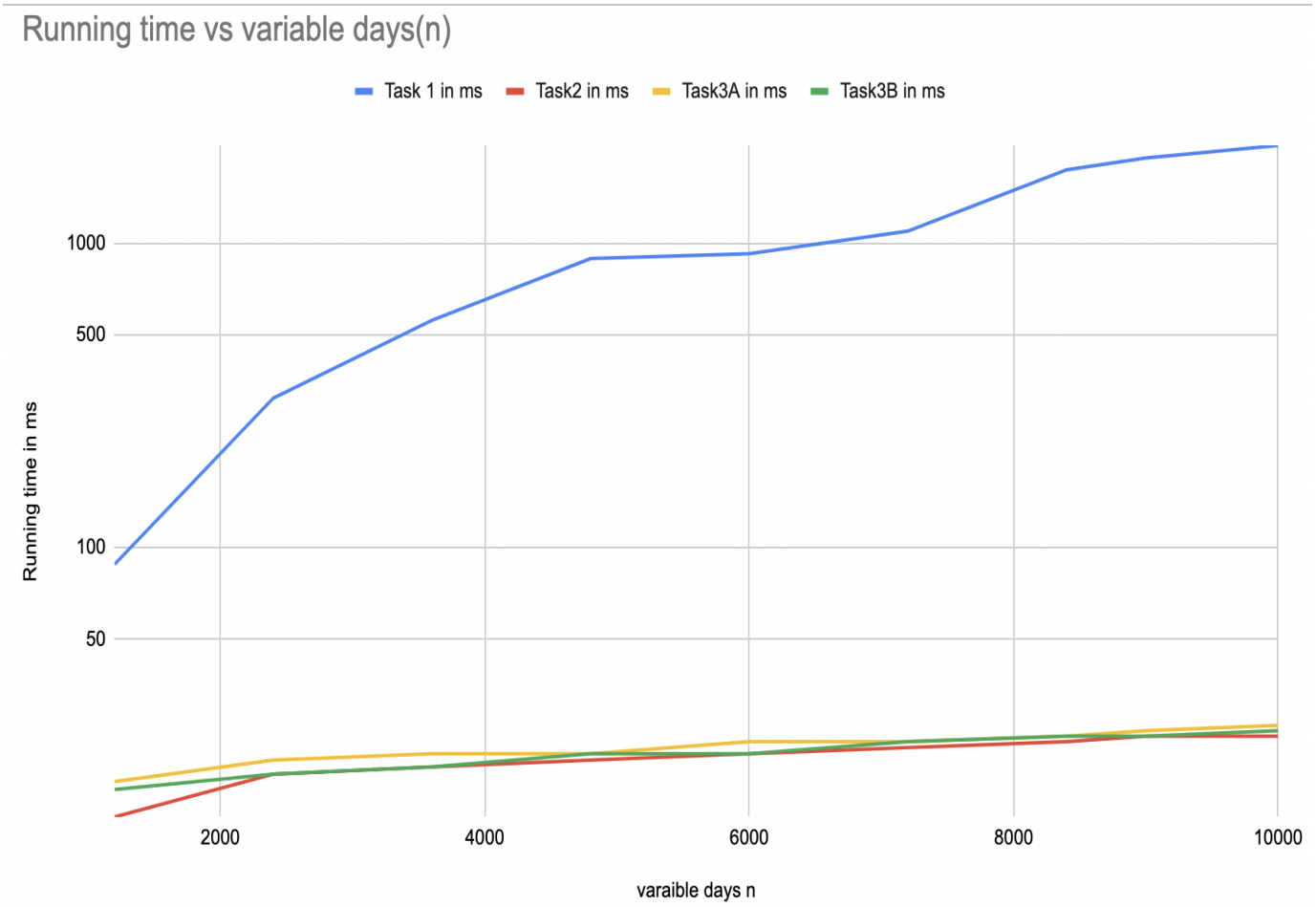
PLOT:1

Problem 1 test cases for constant stocks(m) and variable number of days(n):

No of days(n)	Running time for Task1 in (ms)	Running time for Task2 in (ms)	Running time for Task3A in (ms)	Running time for Task3B in (ms)
1200	88	13	17	16
2400	310	18	20	18
3600	559	19	21	19
4800	893	20	21	21
6000	926	21	23	21
7200	1099	22	23	23
8400	1750	23	24	24
9000	1913	24	25	24
10000	2105	24	26	25

*ms=Milliseconds

Graph plotting for Task1, Task2, Task3A, Task3B Running time vs no of days(n) whereas keeping number of stocks(m) constant.



From the above graph we plotted variable days(n) in X-axis ranging from 1200 to 10000 and the running time in milli seconds as y-axis for Task1 Brute force , Task2 Greedy Algorithm, Task3A Dynamic programming using memoization, Task3B dynamic programming using bottom up approach. We can infer that the running time for brute force is exponentially greater than the other algorithms, and the remaining algorithms will be executing with almost the same running time.

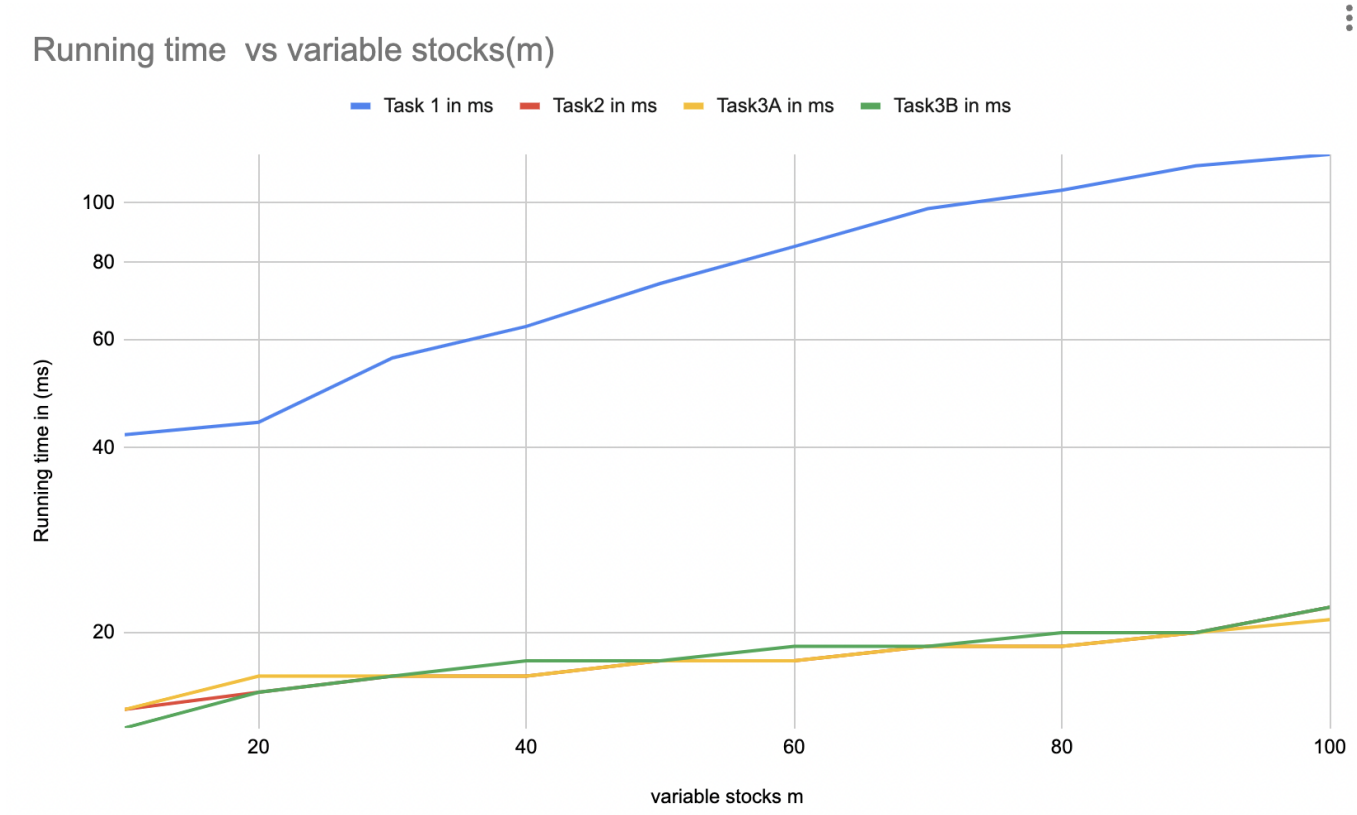
PLOT 2

Problem 1 test cases for constant days(n) and variable number of stocks(m):

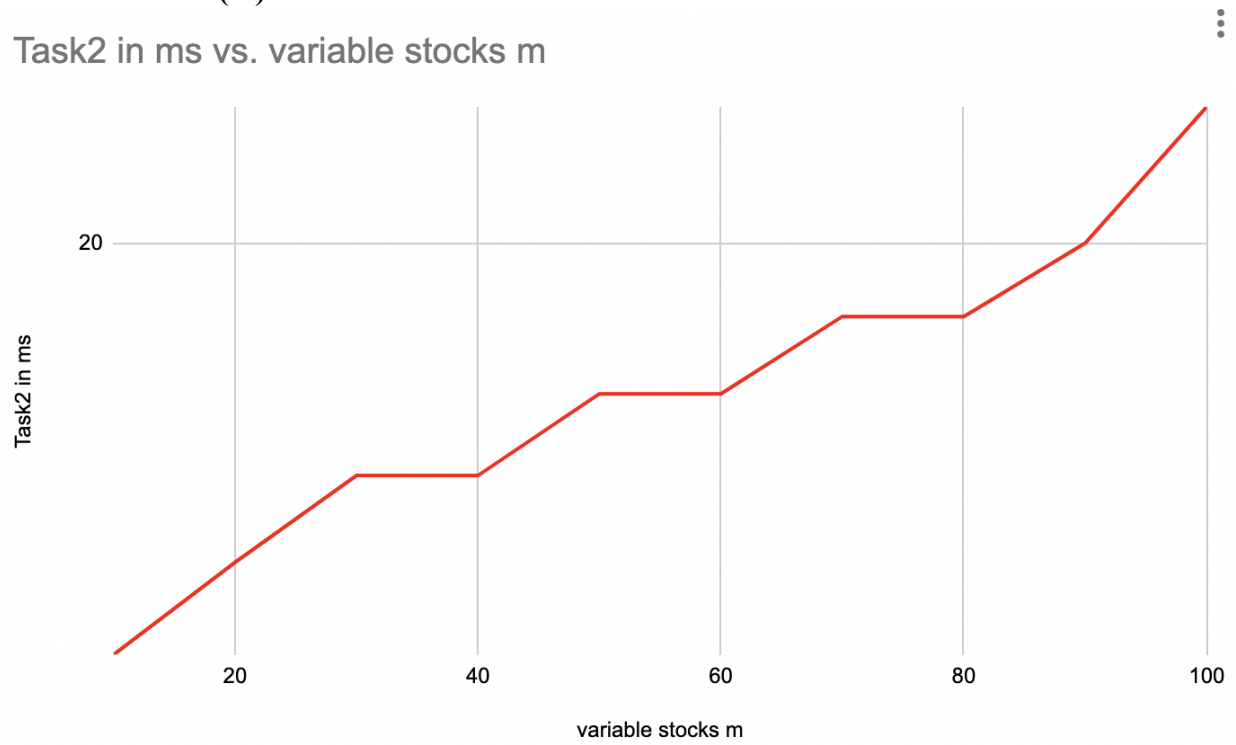
No of stocks(m)	Running time for Task1 in (ms)	Running time for Task2 in (ms)	Running time for Task3A in (ms)	Running time for Task3B in (ms)
10	42	15	15	14
20	44	16	17	16
30	56	17	17	17
40	63	17	17	18
50	74	18	18	18
60	85	18	18	19
70	98	19	19	19
80	105	19	19	20
90	115	20	20	20
100	120	22	21	22

*ms=Milli seconds

Graph plotting for Task1, Task2, Task3A, Task3B Running time v/s no of stocks(n) whereas keeping number of days(n) constant.



Graph plot for Task2 v/s Running time with constant number of days(n) and variable number of stocks(m).



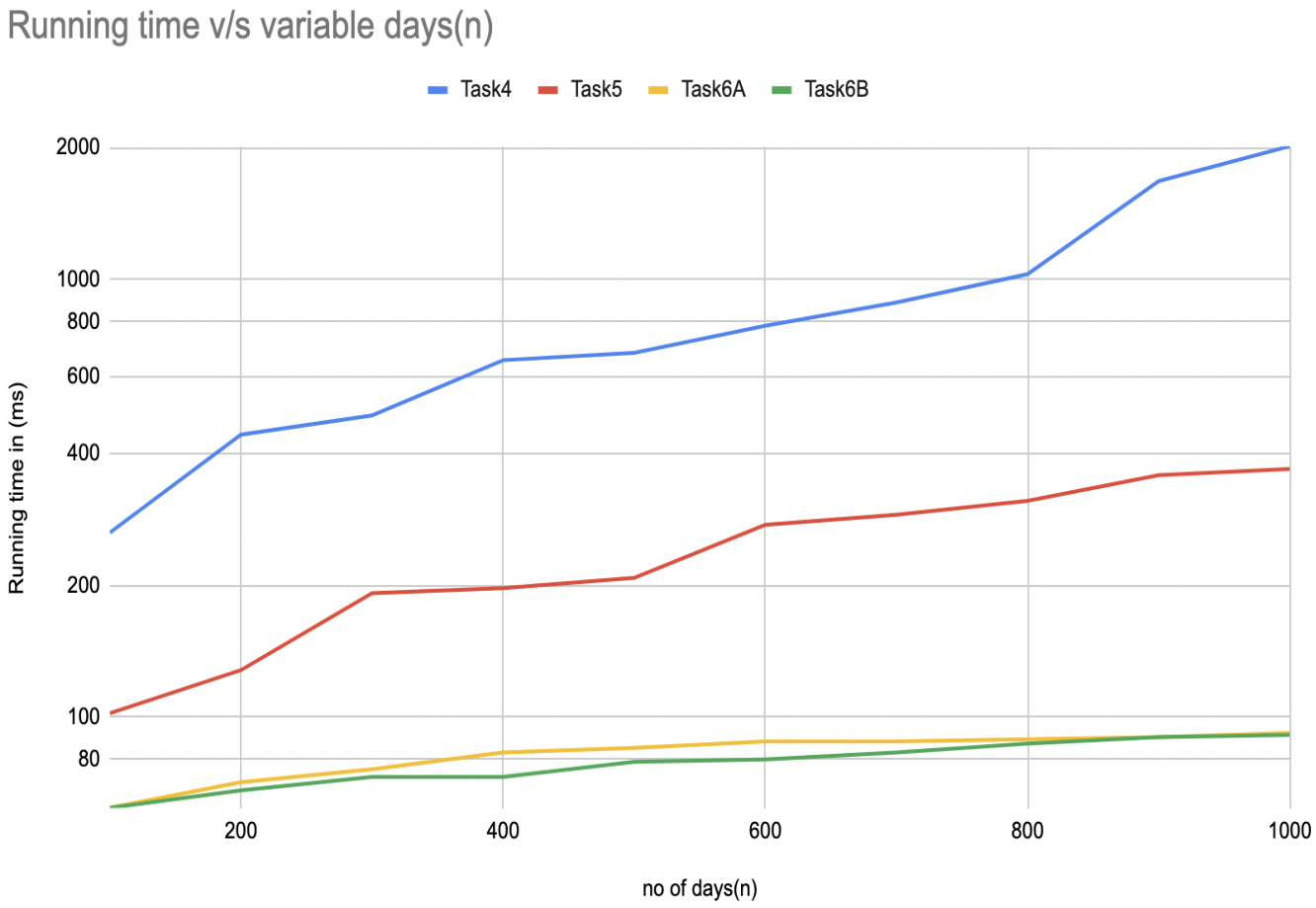
From the above graph we plotted variable stocks(m) in X-axis ranging from 10 to 100, and the running time in milli seconds as y-axis for Task1 brute force algorithm, Task2 greedy algorithm , Task3A dynamic programming using memoization, Task3B dynamic programming using t. We can infer that the running time for brute force is exponentially greater than the other algorithms, and the remaining algorithms will be executing with almost the same running time. As the greedy algorithms have coincided with the dynamic programming plotting, we have placed separate graph plotting for greedy algorithm. So the plottings for greedy and dynamic programming are almost the same level, as there are little fluctuations for memoization and bottom up approaches.

PLOT 3

Problem 2 test cases for constant stocks and transactions(m,k) and variable number of days(n):

No of days(n)	Running time for Task4 in (ms)	Running time for Task5 in (ms)	Running time for Task6A in (ms)	Running time for Task6B in (ms)
100	264	102	62	62
200	442	128	71	68
300	489	192	76	73
400	654	197	83	73
500	680	208	85	79
600	784	275	88	80
700	886	290	88	83
800	1028	312	89	87
900	1678	357	90	90
1000	2016	369	92	91

Graph plot for Task4,Task5,Task6A,Task6B v/s Running time with constant number of stocks and transactions(m,k) and variable number of days(m).



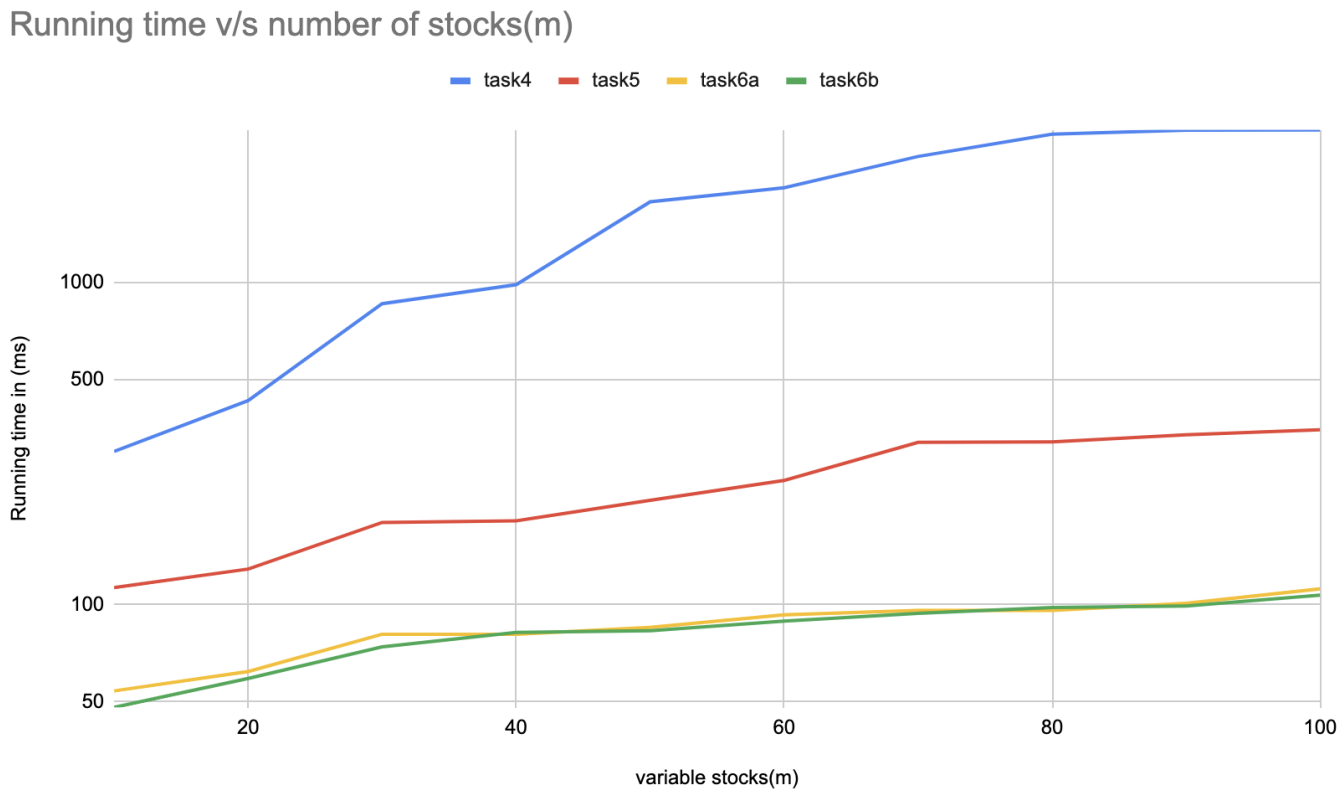
In the above graph we have compare Brute force(Task4), Greedy algorithm(Task5), Dynamic programming using memoization(Task6A), Dynamic Programming using bottom up approach(Task6B) with Running time in milliseconds(ms) keeping number of transactions(k=60), and number of stocks (m=40) constant and changing number of days(from 100 to 1000). From the above graph we can infer that the brute force have the highest running time for each day, which is almost double or exponential for every other algorithms, where as the greedy also have good amount of execution time, however the dynamic programming algorithms both have almost the same time, which is very low which shows that these algorithms are more efficient.

PLOT 4

Problem 2 test cases for constant days and transactions(n,k) and variable number of stocks(m):

No of stocks (m)	Running time for Task4 in (ms)	Running time for Task5 in (ms)	Running time for Task6A in (ms)	Running time for Task6B in (ms)
10	299	113	54	48
20	430	129	62	59
30	860	180	81	74
40	985	182	81	82
50	1782	211	85	83
60	1968	243	93	89
70	2463	319	96	94
80	2887	320	96	98
90	2972	337	101	99
100	2978	349	112	107

Graph plot for Task4,Task5,Task6A,Task6B v/s Running time with constant number of stocks and number of transactions(k,n) and variable stocks(k).



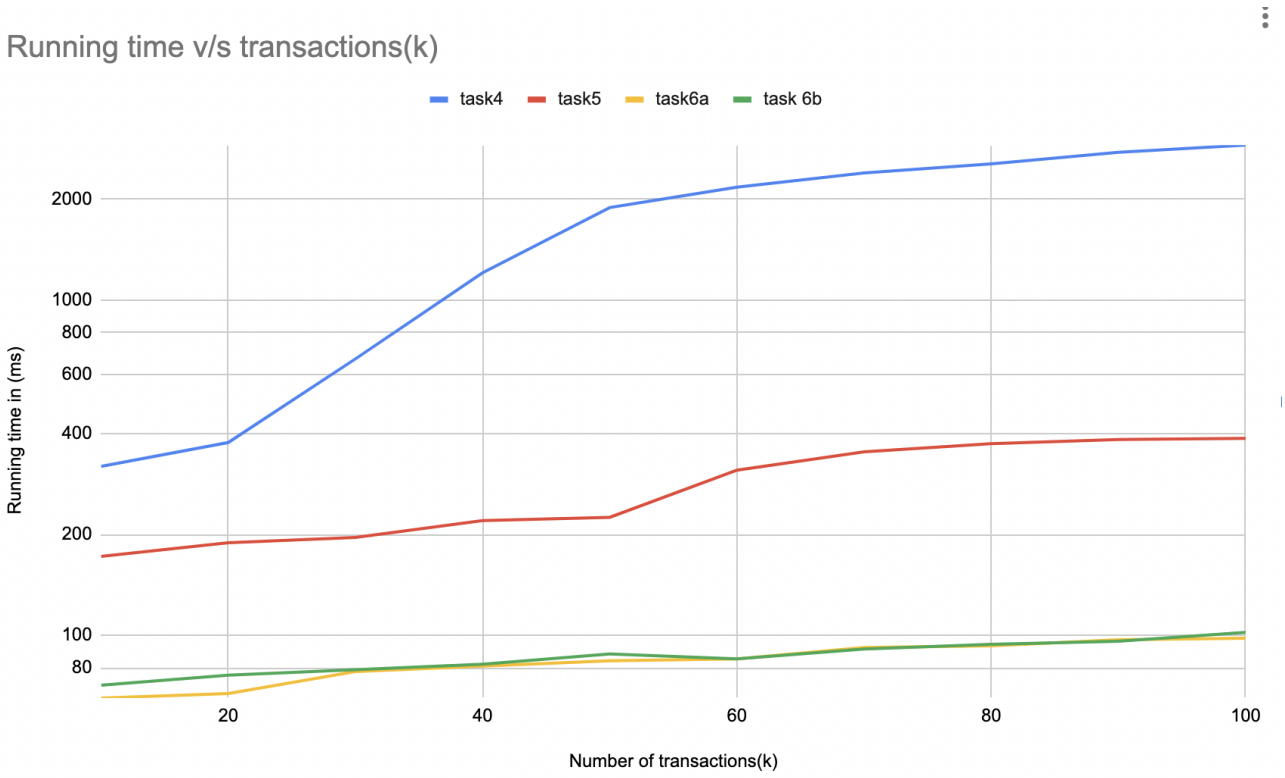
In the above graph we have compare Brute force(Task4), Greedy algorithm(Task5), Dynamic programming using memoization(Task6A), Dynamic Programming using bottom up approach(Task6B) with Running time in milliseconds(ms) keeping number of transactions(k=60), and number of days (n=400) constant and changing number of stocks(from 10 to 100). From the above graph we can infer that the brute force have the highest running time for each day, which is almost double or exponential for every other algorithms, where as the greedy also have good amount of execution time, however the dynamic programming algorithms both have almost the same time, which is very low which shows that these algorithms are more efficient.

PLOT 5

Problem 2 test cases for constant stocks and days(m,n) and variable transactions of stocks(k):

No of Transactions(k)	Running time for Task4 in (ms)	Running time for Task5 in (ms)	Running time for Task6A in (ms)	Running time for Task6B in (ms)
10	319	172	65	71
20	376	189	67	76
30	668	196	78	79
40	1207	220	81	82
50	1890	225	84	88
60	2173	311	85	85
70	2395	353	92	91
80	2550	373	93	94
90	2763	384	97	96
100	2902	387	98	102

Graph plot for Task4,Task5,Task6A,Task6B v/s Running time with constant number of stocks and number of days(m,n) and variable transactions(k).



In the above graph we have compare Brute force(Task4), Greedy algorithm(Task5), Dynamic programming using memoization(Task6A), Dynamic Programming using bottom up approach(Task6B) with Running time in milliseconds(ms) keeping number of stocks(m=40), and number of days (n=40) constant and changing number of transactions(from 10 to 100). From the above graph we can infer that the brute force have the highest running time for each day, which is almost double or exponential for every other algorithms, where as the greedy also have good amount of execution time, however the dynamic programming algorithms both have almost the same time, which is very low which shows that these algorithms are more efficient.

4.Conclusions

This project gave us hands-on experience on how to develop and optimize a solution for a problem. We also got to see visually the difference optimization makes.

Task 1: The brute force implementation with a time complexity of O(m*n^2) is trivial but not efficient as it has to check all the possible scenarios to decide the maximum profit.

Task 2: The greedy approach with a time complexity of O(m*n) is efficient and easy to understand. This approach can be a great solution for not so complex problems.

Task 3A: Even though this is an efficient approach, it has higher space complexity O(n) and readability is less. The time complexity for this algorithm is O(m*n).

Task 3B: Even though this method requires some extra space O(n), it is intuitive and understandable.

Task 4: This is not an efficient approach to solve problem 2 as its time complexity is very high which will have scalability problems and it is hard to implement.

Task 5: This is a good way to implement problem 2. Even though the time complexity is not the best, filling the bottom up table is intuitive and easy to understand.

Task 6A: So far, this has been the most complex algorithm in the project. It is very efficient but not readable. This algorithm also uses extra space but the time complexity is low.

Task 6B: This is a great optimization to task 5 and the most efficient and elegant algorithm that is readable, has low time complexity , uses only some space O(m*n).