COT5405 - Analysis of Algorithms
Programming Project

Team Members -
Nandani Yadav - (1818-6343)
Shaanya Singh - (3476-2752)
Shashi Shirupa - (2572-3497)

Algorithm Design & Analysis:

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

Initialize 2d array (vector in c++)
Take m and n inputs, m- stocks, n- # of days
Take m lines each of n integers and store in the prices array in the form of a .
function-> maxProfit(prices)
Initialize buyDay =0, sellDay = 0, stock = 0, maxprofit = 0

```
for(i=0 to m) -> looping for m stocks
    for(j=0 to n-1) -> looping through each stocks day
        for(k = j + 1 to n)
            currprofit = prices[i][k] - prices[i][j]
            if(currprofit > maxprofit)
                Update maxprofit = currprofit
                Stock = i, buyDay = j, sellDay = k
            End if
        End forLoop (k)
    End forLoop(j)
End forLoop(i)
Print the values (stock, buyDay, sellDay)
```

Time Complexity & Space Complexity-

In this algorithm we have loops that repeat-

- m times (first loop)
- n times (second loop)
- n times (third loop)

Overall the **time complexity** of the algorithm is **O(m*n^2).**

The overall **space complexity** of this algorithm is **O(1).**

Proof of Correctness-

Initialization (outer loop):
Before the loop iterates for the first time, the loop variable has a value of 0. The profit, buy day index, sell day index are all defined to be 0: buyDay =0, sellDay = 0, stock = 0, maxprofit = 0. Therefore, the invariant initially holds true.
Maintenance (outer loop):
Suppose the invariant holds true at the beginning of a certain iteration. The first iteration occurs: 0<=i<=m. Let buyDay, sellDay and maxprofit denote the values at the end of the iteration.
Initialization (middle loop):
Before the loop iterates for the first time, the loop variable has a value of 0. The profit, buy day index, sell day index are all defined to be 0: buyDay =0, sellDay = 0, stock = 0, maxprofit = 0. Therefore, the invariant initially holds true.
Maintenance (middle loop):
Suppose the invariant holds true at the beginning of a certain iteration. The first iteration occurs: 0<=j<=n-1, looping through days of each stock. Let buyDay, sellDay and maxprofit denote the values at the end of the iteration.
Initialization (inner loop):
Before the loop iterates for the first time, the loop variable has a value of 0. The profit, buy day index, sell day index are all defined to be 0: buyDay =0, sellDay = 0, stock = 0, maxprofit = 0. Therefore, the invariant initially holds true.
Maintenance (inner loop):
Suppose the invariant holds true at the beginning of a certain iteration. The first iteration occurs: j+1<=k<=n, looping through the days after the current stock and so on. Let buyDay, sellDay and maxprofit denote the values at the end of the iteration.

Initialize currprofit
currprofit = prices[i][k] - prices[i][j]
if(currprofit > maxprofit):
Update maxprofit = currprofit
Stock = i, buyDay = j, sellDay = k
End if

This gives us the maxprofit as well as transaction indices.
Therefore, the invariant holds true at the end of each iteration.
Termination (inner loop):
It terminates when the loop variable is done with running in the range of j+1<=k<=n.
Termination (middle loop):
It terminates when the loop variable is done with running in the range of 0<=j<=n-1.
Termination (outer loop):
It terminates when the loop variable is done with running in the range of 0<=i<=m.
Correctness: The loops terminate and give us maxprofit, buyDay and sellDay.

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

Initialize 2d array (vector in c++)
Take m and n inputs, m- stocks, n- # of days
Take m lines each of n integers and store in the prices array in the form of a vector.
Function -> maxProfit(prices, n):

```
Initialize a 2d vector for storing indices for m stocks
Initialize stock =0, buyDay =0, sellDay = 0
for(i=0 to m)
    Initialize currStock = prices[i][0]
    for(j = 0 to n)
        if(prices[i][j]<currStock)
                currStock = prices[i][j]
                 stock = i;
                buyDay = j;
            End if
            if((prices[i][j] - currStock) > maximum[i][0])
                maximum[i][0] = prices[i][j] - currStock;
                sellDay = j;
                maximum[i][1] = stock;
                maximum[i][2] = buyDay;
                maximum[i][3] = sellDay;
            End if
      End forLoop (j)
End forLoop(i)
Print(stock, buyDay, sellDay)
```

Time Complexity & Space Complexity-

In this algorithm we have loops that repeat-

- m times (first loop)
- n times (second loop)

Overall the **time complexity** of the algorithm is **O(m*n).**

The overall **space complexity** of this algorithm is **O(1).**

Proof of Correctness-

Initialization (outer loop):

Before the loop iterates for the first time, loop variable has a value of 0. The profit, buy day index, sell day index are all defined to be 0: buyDay =0, sellDay = 0, stock = 0, maxprofit = 0. Therefore, the invariant initially holds true.

Maintenance (outer loop): Suppose the invariant holds true at the beginning of a certain iteration. The first iteration occurs: 0<=i<=m (iterating over stocks). Let buyDay, sellDay and maxprofit denote the values at the end of the iteration.

Initialize currStock = prices[i][0]

Initialization (inner loop):
Before the loop iterates for the first time, loop variable has a value of 0. The profit, buy day index, sell day index are all defined to be 0: buyDay =0, sellDay = 0, stock = 0, maxprofit = 0. Therefore, the invariant initially holds true.

Maintenance (inner loop):
Suppose the invariant holds true at the beginning of a certain iteration. The first iteration occurs: 0<=j<=n (iterating over each day in each stock). Let buyDay, sellDay and maxprofit denote the values at the end of the iteration.

```
if(prices[i][j]<currStock)
currStock = prices[i][j]
             stock = i;
            buyDay = j;
End if
if((prices[i][j] - currStock) > maximum[i][0])
maximum[i][0] = prices[i][j] - currStock;
             sellDay = j;
            maximum[i][1] = stock;
           maximum[i][2] = buyDay;
          maximum[i][3] = sellDay;
End if
```

Termination (inner loop):
It terminates when the loop variable is done with running in the range of 0<=j<=n and stores updated values for stock, buyDay and sellDay.

Termination (outer loop):
It terminates when the loop variable is done with running in the range of 0<=i<=m.
Correctness: The loops terminate and give us maxprofit, buyDay and sellDay.

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

Top Down DP:

Initialize 2d array (vector in c++)
Take m and n inputs, m- stocks, n- # of days
Take m lines each of n integers and store in the prices array in the form of a vector.
Function -> findMax(prices,m,n)
Declare a profit[100] array
Initialize tempCheapest = 0, count = 0, sell = -1, buy = -1, stock = -1, overAllMax

```
if(n==1)
        profit[n]=max(0,profit[n]=prices[n]-prices[n-1])
        if(profit[n]>overAllMax)
            buy=n-1
        End if
         overAllMax=max(overAllMax,profit[n]);
         if(profit[n]==0)
            tempCheapest = n
         Else
            tempCheapest = 0
         End if else
         return profit[n]
 else
        profit[n]=max(0,findMax(prices,n-1,m)+prices[n]-prices[n-1])
  if(profit[n]==0)
      tempCheapest = n
   if(profit[n]>overAllMax)
         buy=tempCheapest;
          sell=n;
          stock=m;
   End if
    overAllMax=max(overAllMax,profit[n])

   return profit[n]
   Print(stock, buy , sell)
```

Time Complexity & Space Complexity-

Overall the **time complexity** of the algorithm is **O(m*n).**

The overall **space complexity** of this algorithm is **O(n).**

Recursive Formulation of Optimal Substructure

$$
\text{profit}[n] = \begin{cases} \text{if } n == 1, \\ \text{profit}[n] = \max(0,\ \text{profit}[n] = \text{prices}[n] - \text{prices}[n-1]) \\ \text{else}, \\ \text{profit}[n] = \max(0,\ \text{findMax}(\text{prices}, n-1, m) + \\ \qquad \text{prices}[n] - \text{prices}[n-1])) \end{cases}
$$

Bottom Up DP:

Initialize 2d array (vector in c++)
Take m and n inputs, m- stocks, n- # of days
Take m lines each of n integers and store in the prices array in the form of a vector.

Function -> find(prices,m,n)

```
Initialize buyDay = 1, sellDay = 1,  maxTotal = 0, maxprofit = 0, buyMin =1 , sellMax = 1 , stock =
0
Initialize profit[n] array
Profit[0] = 0
for(j=0 to m)
   for(i = 1 to n)
       Profit[i] = max(0, profit[i-1]+(prices[j][i] - prices[j][i-1]))
       if((prices[j][i] - prices[j][i-1] < 0) && profit[i] == 0)
         buyDay = i+1
       if(maxtotal > profit[i])
         sellDay = i+1
       End if
       maxtotal = max(maxtotal, profit[i])
       if(maxtotal > maxprofit)
         buyMin = buyDay
         sellMax = sellDay
         Stock = j+1
       End if
       Maxprofit = max(maxprofit, maxtotal)
    End forLoop (i)
End forLoop(j)
print(stock, buyMin, sellMax)
```

Time Complexity & Space Complexity-

In this algorithm we have loops that repeat-

- m times (first loop)
- n times (second loop)

Overall the **time complexity** of the algorithm is **O(m*n).**

The overall **space complexity** of this algorithm is **O(n).**

Recursive Formulation of Optimal Substructure

$$profit[i] = \max \begin{cases} 0 \\ profit[i-1] + prices[j][i] - prices[j][i-1] \end{cases}$$

Proof of Correctness-

Base Case:
For the first day the profit array will have a value of 0.

Induction Hypothesis:
The algorithm takes an input of m stocks which have values for n days and dynamically updates the maximum profit for day i.

Inductive Step:
The algorithm is able to calculate the maximum profit achieved for m = 1. As the number of stocks increase, it is able to dynamically calculate the profit among the m stocks and update the profit array with the maximum yielded profit for the certain day.

Profit[i] = max(0, profit[i-1]+(prices[j][i] - prices[j][i-1]))

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

Declare a class

Declare public variables: maxProfit, output

Declare a class MaxProfit

Function -> maxprofit(prices,day,k,isBuy,stock,buyDay,output)

Initialize maxProfit to 0

Declare finalOutput variable as 2D vector

If (day >= number of days (n) or if k is 0)

Then return {maxProfit,output}

End if

If (isBuy is true)

Hold the stock

Function call: maxprofit(prices,day + 1, k, isBuy, stock, buyDay, output)

Initialize profit > maxProfit = maxAns.maxProfit

Initialize currentOutput = maxAns.output

If (profit > maxProfit)

Then maxProfit = profit

FinalOutput = currentOutput

Sell the stock

Initialize diff = prices[stock][day] - prices[stock][buyDay]

Initialize modifiedOutput = output

Initialize 1D vector temp

Insert stock, buyDay and day in temp

Push back temp in modifiedOutput

Function call: maxprofit(prices, day, k - 1, false, 0, 0, modifiedOutput) into maxAns

Update profit and currentOutput values

End if

Else

Skip the day

Initialize outputcopy = output

Function call: maxprofit(prices, day + 1, k, false, 0, 0, outputcopy) into helperRes

Update profit = helperRes.maxProfit

currentOutput = helperRes.output

If (profit>maxProfit)

Then update maxProfit = profit

finalOutput = currentOutput

End if

Buy stock of one of the companies

Iterate number of stocks: for (int i=0; i<prices.size(); i++)

Function Call: maxprofit(prices, day + 1, k, true, i, day, outputcopy) *into helperRes*

Update profit = helperRest.maxProfit and currentOuput = helperRes.output
If (profit > maxProfit)
Then update maxProfit to profit and finalOutput to currentOutput
End if
End for
End else

Return maxProfit and finalOuptut

<u>Time Complexity & Space Complexity-</u>

Overall the **time complexity** of the algorithm is **O(m*n^2k).**

The overall **space complexity** of this algorithm is **O(1).**

<u>Proof of Correctness-</u>
Base case:
k = 1 (for one transaction)
To prove: P(1) holds true.
We have a 2D vector ‘prices’ (mxn) where m corresponds to the number of stocks and n corresponds to the days. prices[m][n] gives us the price of the mth stock on the nth day.
For k = 1
We perform multiple transactions and keep updating the maxProfit as well as the buy and sell dates. For 1 transaction we choose the transaction that gives us the max profit after comparing it with every other transaction.

Induction Hypothesis:
Assume the algorithm holds true for k transactions i.e, for m stocks over n days, we get the k best transactions that do not collide with each other.

Inductive Step:

If the algorithm holds true for k transactions it should also hold true for k+1 transactions.

To prove: P(k+1) also holds true. All values of profits are compared over all the stocks to find the best transaction values. We create an array when we buy a stock and as well as when we sell a stock. We choose the k+1 best transactions for the inputs given. This is updated each time the maxprofit function is called.

```
maxAns= maxprofit(prices, day, k - 1, false, 0, 0, modifiedOutput);
 profit = maxAns.maxProfit;
 currentOutput = maxAns.output;
 profit += diff;
 if(profit > maxProfit) {
     maxProfit = profit;
     finalOutput = currentOutput;
}
```

Output is returned as: {maxProfit, finalOutput}

Alg5 Design a Θ(m ∗n2∗k) time dynamic programming algorithm for solving Problem2

```
Initialize 2d array (vector in c++)
Take input for k- number of transactions
Take m and n inputs, m- stocks, n- # of days
Take m lines each of n integers and store in the prices array in the form of a vector.
Function -> find(prices,k)
Initialize profits[k+1][n] of all 0's
for(t=1 to k+1)
   for(m=0 to size(prices))
      for(j=1 to size(prices[0]))
         for(l =0 to j)
            profits[t][j] = max(profits[t][j], profits[t][j-1], prices[m][j]+profits[t-1][l]-prices[m][l])
         End forLoop l
      End forLoop j
   End forLoop m
End forLoop t

Initialize vector days, i = profit.size() - 1,  j = profit[0].size() - 1
while(True)
   Initialize counter = 0
   If (i==0 or j ==0)
     break
   if (profit[i][j] == profit[i][j-1])
     j = j - 1
   else
      days.add(j+1)
      for(int l=0 to prices.size())
        maxDiff = profit[i][j] - prices[l][j]
         for(int k = j-1 to 0, k = k -1)
            if (profit[i-1][k] - prices[l][k] == maxDiff)
                i = i - 1
                j = k
                days.add(j+1)
                days.add(l+1)
                flag=1
            End if
          if flag == 1 break
        End forLoop(k)
      if flag == 1 break
   End Else
End whileLoop

Initialize stock, buyDay, sellDay,  c = size(days)/3
```

```
while c > 0
    Stock = days.back()
    days.pop()
    buyDay= days.back()
    days.pop()
    sellDay= days.back()
    days.pop()
    c = c -1
    print(stock, buyDay, sellDay)
```

Time Complexity & Space Complexity-

In this algorithm we have loops that repeat-

- k+1 times (first loop)
- m times (second loop)
- n times (third loop)
- n-1 times (fourth loop)

Overall the **time complexity** of the algorithm is **O(k*m*n^2).**

The profit table of dimensions k*n is being updated therefore, the overall **space complexity** of this algorithm is **O(k*n).**

Recursive Formulation of Optimal Substructure

$$\text{profit}[t][j] = \max \begin{cases} 0 & t \text{ or } j = 0 \\ \text{profit}[t][j-1] & \\ \max\limits_{\substack{0 \le l \\ l < j}} \left(\text{prices}[m][j] + \text{profit}[t-1][l] - \text{prices}[m][l]\right) & \end{cases}$$

Proof of Correctness-

Base Case:

For k = 0 or n = 0 the profit on that transaction is 0 as there is no previous transaction for k=0 that has been done to yield a profit and no stock sold on n = 0 to yield profit.

Induction Hypothesis:

The profit table dynamically updates the maximum profit possible for kth transaction on nth day. For the kth transaction the table takes the previous transaction and then calculates the maximum possible profit until (n-1)th day. It then uses this maximum possible profit and adds it with the price on day n.

Induction:
For k =1 the algorithm is able to calculate the maximum profit among ‘m’ number of stocks. The algorithm iteratively calculates the maximum profit gained from the previous transaction till (n-1)th day and adds it to the current day (day n) price. This process is done iteratively for k+1 times and then maximum profit is retrieved.

```
for(t=1 to k+1)
   for(m=0 to size(prices))
      for(j=1 to size(prices[0]))
         for(l =0 to j)
            profits[t][j] = max(profits[t][j], profits[t][j-1], prices[m][j]+profits[t-1][l]-prices[m][l])
         End forLoop l
      End forLoop j
   End forLoop m
End forLoop t
```

The algorithm is able to give the order of transactions for the maximum profit for any k value.

Alg6 Design a Θ(m ∗n ∗k) time dynamic programming algorithm for solving Problem2

Top Down DP:

```
Initialize 2d array (vector in c++)
Take input for k- number of transactions
Take m and n inputs, m- stocks, n- # of days
Take m lines each of n integers and store in the prices array in the form of a vector.
Function -> find(prices,t,j)
if(t==0 or j==0)
    return 0
for(m=0 to t+1)
   mtf = INT_MIN
   mtf = max(mtf, maxprofit(prices,t-1,j-1)-prices[m][j-1])
   profit[t][j]=max(max(prices[m][j]+mtf , maxprofit(prices,t,j-1)), profit[t][j])
End forLoop(m)
Initialize vector days, i = profit.size() - 1,  j = profit[0].size() - 1
while(True)
    Initialize counter = 0
    If (i==0 or j ==0)
       break
    if (profit[i][j] == profit[i][j-1])
       j = j - 1
    else
        days.add(j+1)
        for(l=0 to prices.size())
           maxDiff = profit[i][j] - prices[l][j]
            for(int k = j-1 to 0, k = k -1)
               if (profit[i-1][k] - prices[l][k] == maxDiff)
                    i = i - 1
                    j = k
                    days.add(j+1)
                    days.add(l+1)
                    flag=1
                End if
              if flag == 1 break
           End forLoop(k)
        if flag == 1 break
    End Else
End whileLoop


Initialize stock, buyDay, sellDay,  c = size(days)/3
while c > 0
    Stock = days.back()
```

days.pop()
buyDay= days.back()
days.pop()
sellDay= days.back()
days.pop()
c = c -1
print(stock, buyDay, sellDay)

<u>Time Complexity & Space Complexity-</u>

The **time complexity** of the algorithm is **O(k*m*n).**

The profit table of dimensions k*n is being updated therefore, the overall **space complexity** of this algorithm is **O(k*n).**

<u>Recursive Formulation of Optimal Substructure</u>

$$profit[t][j] = \max \begin{cases} \text{if } t \text{ or } j == 0, \\ \quad 0 \\ \text{else,} \\ \max(prices[m][j] + mtf,\ maxprofit(prices, t, j-1)) \\ profit[t][j] \end{cases}$$

where,

$$mtf = \max \begin{cases} mtf \\ maxprofit(prices, t-1, j-1) - prices[m][j-1] \end{cases}$$

<u>Bottom Up DP:</u>

```
Initialize 2d array (vector in c++)
Take input for k- number of transactions
Take m and n inputs, m- stocks, n- # of days
Take m lines each of n integers and store in the prices array in the form of a vector.
```

Function -> find(prices,k)

```
Initialize profits[k+1][n] of all 0's
for(t = 1 to k+1)
Initialize mtf(m, INT_MIN)
   for(m=0 to size(prices))
     for(j=1 to size(prices[0]))
       mtf[m] = max(mtf[m], profit[i-1][j-1] - prices[m][j-1])
       Profit[t][j] = max(profit[t][j],profit[t][j-1],prices[m][j] + mtf[m])
      End forLoop(j)
  End forLoop(m)
End forLoop(t)

Initialize vector days, i = profit.size() - 1,  j = profit[0].size() - 1
while(True)
   Initialize counter = 0
   If (i==0 or j ==0)
     break
   if (profit[i][j] == profit[i][j-1])
     j = j - 1
   else
      days.add(j+1)
      for(l=0 to prices.size())
        maxDiff = profit[i][j] - prices[l][j]
        for(int k = j-1 to 0, k = k -1)
          if (profit[i-1][k] - prices[l][k] == maxDiff)
              i = i - 1
              j = k
              days.add(j+1)
              days.add(l+1)
              flag=1
           End if
          if flag == 1 break
        End forLoop(k)
      if flag == 1 break
   End Else
End whileLoop


Initialize stock, buyDay, sellDay,  c = size(days)/3
```

```
while c > 0
    Stock = days.back()
    days.pop()
    buyDay= days.back()
    days.pop()
    sellDay= days.back()
    days.pop()
    c = c -1
    print(stock, buyDay, sellDay)
```

Time Complexity & Space Complexity-

In this algorithm we have loops that repeat-

- k+1 times (first loop)
- m times (second loop)
- n times (third loop)

Overall the **time complexity** of the algorithm is **O(k*m*n).**

The profit table of dimensions k*n is being updated therefore, the overall **space complexity** of this algorithm is **O(k*n).**

Recursive Formulation of Optimal Substructure

$$profit[t][j] = \max \begin{cases} 0 & t \text{ or } j = 0 \\ profit[t][j-1] & \\ prices[m][j] + mtf[m] & \end{cases}$$

where,

$$mtf[m] = \max \begin{cases} mtf[m] \\ profit[t-1][j-1] - prices[m][j-1] \end{cases}$$

<u>Proof of Correctness-</u>

Base Case:

Similar to algorithm 5 for k = 0 and n = 0 the profit yielded is 0.

Induction Hypothesis:

The profit table dynamically updates the maximum profit possible for kth transaction on nth day. For the kth transaction the table takes the previous transaction and uses the maximum possible profit already stored and adds it with the price on day n.

Induction:

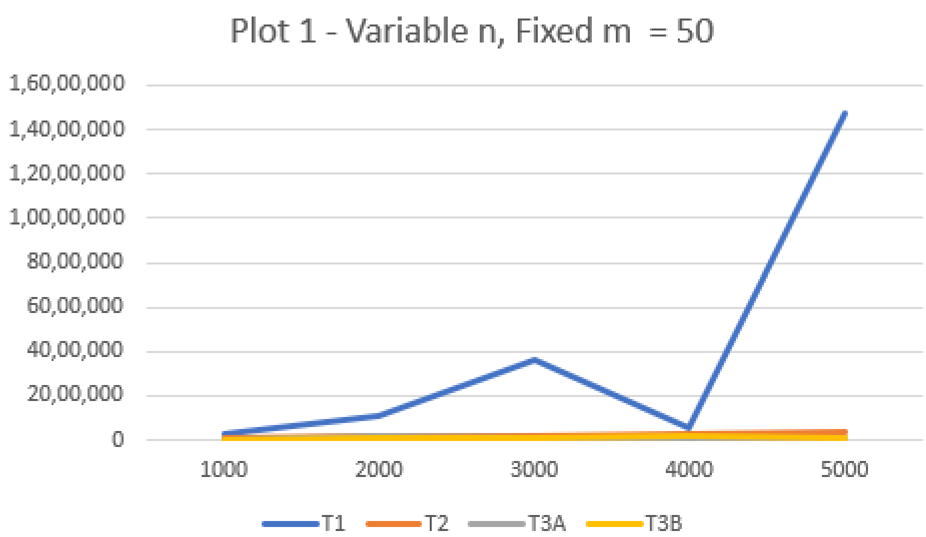
For k =1 the algorithm is able to calculate the maximum profit among ‘m’ number of stocks. The algorithm calculates the maximum profit gained from the previous transaction till which is constantly updated in a variable and then adds it to the current day (day n) price. This process is done iteratively for k+1 times and then maximum profit is retrieved.

```
for(t = 1 to k+1)
Initialize mtf(m, INT_MIN)
   for(m=0 to size(prices))
      for(j=1 to size(prices[0]))
         mtf[m] = max(mtf[m], profit[i-1][j-1] - prices[m][j-1])
         Profit[t][j] = max(profit[t][j],profit[t][j-1],prices[m][j] + mtf[m])
        End forLoop(j)
  End forLoop(m)
End forLoop(t)
```

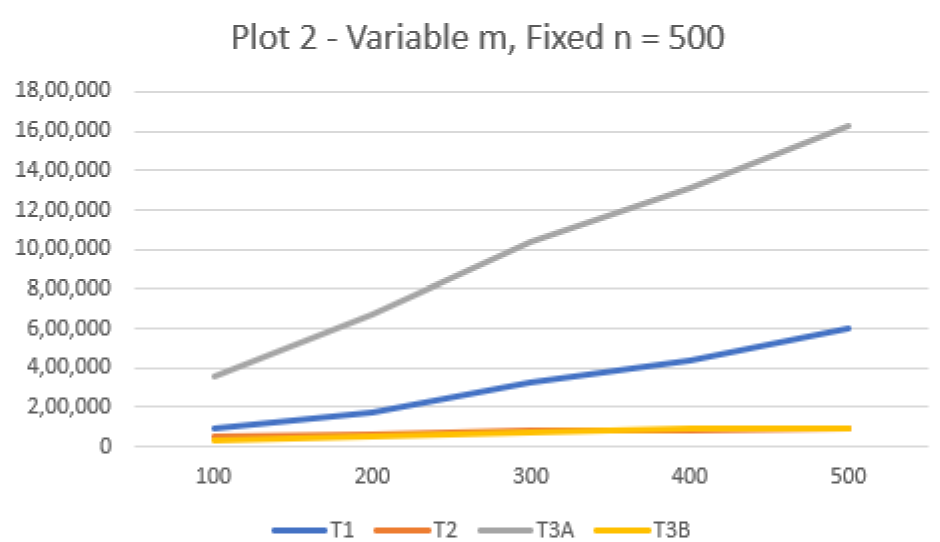
The algorithm is able to give the order of transactions for the maximum profit for any k value.

Experimental Comparative Study:

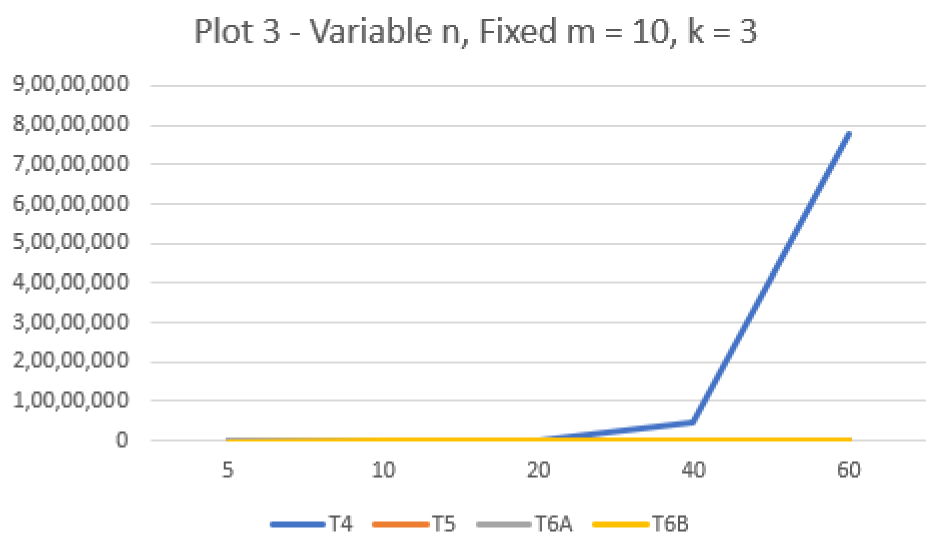
Plot1 Comparison of Task1, Task2, Task3A, Task3B with variable n and fixed m



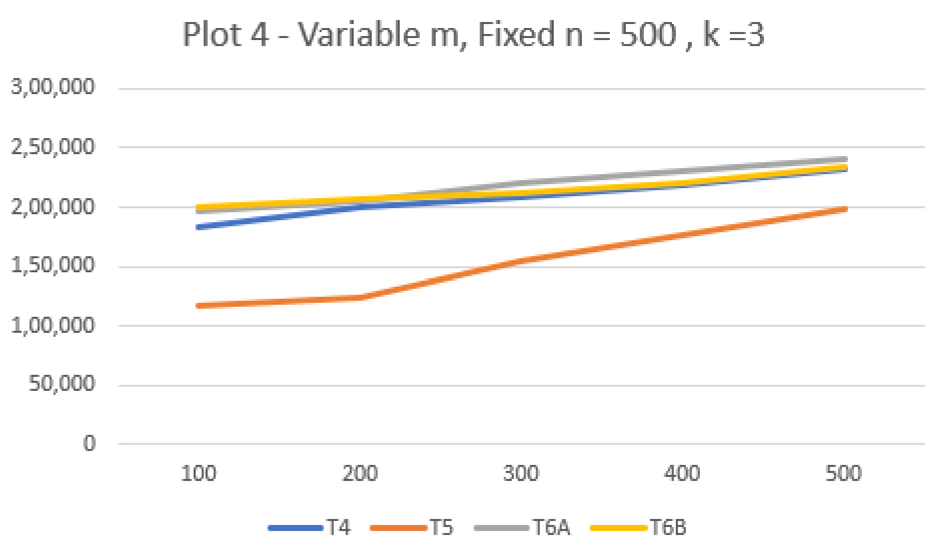
Plot2 Comparison of Task1, Task2, Task3A, Task3B with variable m and fixed n



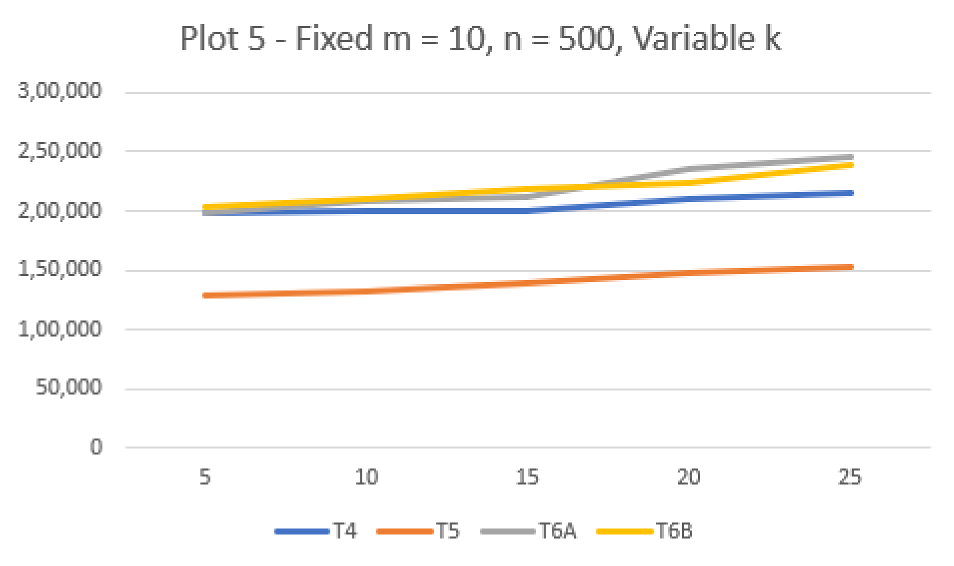
Plot3 Comparison of Task4, Task5, Task6A, Task6B with variable n and fixed m and k



Plot4 Comparison of Task4, Task5, Task6A, Task6B with variable m and fixed n and k



Plot5 Comparison of Task4, Task5, Task6A, Task6B with variable k and fixed m and n



Execution:

Makefile:

```
output: Stocks.o
	g++ Stocks.o -o Stocks
Stocks.o: Stocks.cpp
	g++ -c Stocks.cpp

clean:
	rm *.o Stocks
```

Running tasks:

```
PROBLEMS 1   OUTPUT   DEBUG CONSOLE   TERMINAL   JUPYTER

1 2 3
3 4 5
(base) → AOA ./Stocks 6b < input
3 1 2
1 2 3
3 4 5
(base) → AOA ./Stocks 5 < input
3 1 2
1 2 3
3 4 5
(base) → AOA ./Stocks 5 < input
3 1 2
1 2 3
3 4 5
(base) → AOA clang++ -std=c++11 Stocks.cpp -o Stocks
(base) → AOA ./Stocks 5 < input
3 1 2
1 2 3
3 4 5
(base) → AOA
```

Conclusion:

This project for the class COT5405 - Analysis of Algorithms Programming Project revolved around buying and selling of multiple stocks over multiple days. Our goal in each problem was to find the sequence of transactions that would give us the maximum profit. After understanding the problem statement and creating a baseline approach, we proceeded to the algorithm design. We did this in accordance with the complexities that were asked of us in each case. Application of these algorithms were done in the programming task. Problem 1 where we had to use just one transaction overall was relatively simpler to implement as we did not have to worry about overlapping of transaction dates. One of the most challenging aspects of this project was to implement the backtracking which would give us the sequence of dates on which the buying and selling was done. This was especially tough in the brute force algorithm for k transactions as we were trying to avoid clashing of dates (cannot hold another stock if we already have one in hand). This project allowed us to get a handle on the working and application of dynamic programming as well as designing an algorithm specific to time complexities provided. Specification of time complexities gave us an architectural understanding of code design and allowed us to experiment with different structures and methods to obtain the desired results. All in all this project was a great learning experience which challenged us and allowed us to question our logical thinking as well as algorithmic thinking.

Contribution of each member

Nandani Yadav:
Task2, Task 3A, Task 3B, respective task's algorithm, recurrence relation, time and space complexity, proof of correctness, created makefile

Shaanya Singh:
Task 1, Task 2, Task 4, respective task's algorithm, recurrence relation, time and space complexity, proof of correctness, graph plotting

Shashi Shirupa
Task5, Task 6A, Task 6B, respective task's algorithm, recurrence relation, time and space complexity, proof of correctness