# AOA Programming Project

**Team Members:**

Arshdeep Kaur, arshdeep.kaur@ufl.edu , UFID - 79001937

Gayathri Madala, gmadala@ufl.edu , UFID - 80357003

Krishna Keshav, kkeshav@ufl.edu , UFID - 96642399

## Problem Statement 1:

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

## Algorithm Design Task:

***ALG 1*** - Design a Θ (m $*$n$^2$) time brute force algorithm for Problem 1

### 1.1 Algorithm Description:

We have an m*n matrix as input where each row represents a stock. Each row has a sequence of n elements which represent the price of a single stock in each of n days. If we imagine one row as a 1-D matrix (single stock), we can calculate profit for every transaction for this stock, buying the stock on day $n_j$ *where j ranges from 1 to n* and calculate profit for selling it on all the subsequent days $n_k$ where k = *(j + 1) up to n*.

We find single transaction with maximum profit by continuously tracking profit for each transaction and comparing it with the maximum profit transaction till that point. Whenever we get a greater profit transaction, we save the details for that transaction.

Since we calculate profit for each pair of transaction for one stock, the loop runs n*(n-1) times and thus, the complexity of finding maximum profit transaction for a single stock is O(n$^2$).

We extend this algorithm to find maximum profit of m stocks, by repeating the same for every row of the given 2d- matrix. Since we execute the above algorithm for *m* stocks, the complexity would increase by a factor of m.

The complexity of finding maximum profit transaction for m stocks whose prices are given for n consecutive days is O(m*n$^2$).

## 1.2 Algorithm Design:

1. Use variable to store maximumProfit till any given transaction and initiate it to 0

2. Loop over each stock (1 to m)

3. For each stock, we calculate profit for all possible transactions

4. If profit of Current Transaction is greater than the maximumProfit untill now, we update the maximumProfit value and the transaction details.

Row number represents the stock number, and buy day, sell day are represented by the column indices of the current transaction.

5. Return the transaction details.

**Pseudo Code:-**

```
Maximum_Profit(m*n Array A):

        INITIALIZE maximumProfit = 0
        INITIALIZE Stock Number, Stock buying day, Stock selling day = 0
        FOREACH Stock i= 1 to m:
        FOREACH day we can buy any stock j = 1, 2…. n:
        FOREACH day where we can sell current bought stock k = (j + 1), 2…n:
              CurrentProfit = Profit of Current transaction ( A[i][k] - A[i][j] )
        IF (CurrentProfit > maximumProfit)
        Update maximumProfit = CurrentProfit
        Save the details of current transaction
        Stock Number = i
        Stock buying day = j
        Stock selling day = k
END FOR
     END FOR
   END FOR
                PRINT/RETURN Transaction Details
```

*If all the transaction details are 0, it means there is no transaction which is giving any profit.*

## 1.3 Proof of Correctness:

For the trivial case of only **1 stock and 2 days** –

There is only 1 possible transaction – buy stock on first day and sell on second day.

As per our algorithm, if this transaction gives profit(>0), it will return these transaction details.

For **1 stock and n days** –

We calculate the profit for each possible transaction for buying it on any day and selling it on a later day. The transaction details and maximum profit data is updated only if the current transaction's profit is greater than the maximum profit of any previous transactions. Thus, transaction details always hold the details of the transaction which gave the maximum profit.

Let O be an optimal solution, it will always point to the transaction with maximum profit until any given transaction. Our solution also updates the transaction details only if the new transaction gives more profit than the previous maximum profit. Thus, our solution is optimal as it always stores the transaction that gives maximum profit.

Extending above for **m stocks and n days** – We perform the same process for every stock, and thus the transaction details will be updated only if we get more profit for any transaction of some other stock. Thus, transaction details always store data of the stock which gave maximum profit.

**All the transactions for which the profit is calculated are valid** – Since the day the stock is sold is always greater than the day it was bought, the transactions are always valid.

**The algorithm terminates -** Since we loop only for m stocks and for each stock, we loop only for valid transactions up to $n^{th}$ day. Thus, the algorithm will always terminate as it has limited valid transactions and we calculate profit only of the valid transactions.

## 1.4 Complexity:

The Algorithm Description stated above, provides a justification for the complexities stated below

- Time complexity: $O(m*n^2)$ = Loop runs $(m*n*(n-1))/2$ times
- Space complexity: O(1) = Because only few variables - maximumProfit, currentProfit, stock, buyDay, sellDay are used. We do not use any additional memory that depends on the input size.

==========================================================================

**Algorithm Design Task:**

***ALG 2*** - Design a $\Theta(m * n)$ time greedy algorithm for solving Problem1

## 2.1 Algorithm Description:

We simplify the problem to single stock(for 1 row of given m*n matrix) which will narrow down the problem to 1-d array and then repeat those for each of the m stocks.

Let P be an array which will denote price prediction of one stock for n days. The idea is that the maximum profit is basically a maximum difference between 2 elements on day j and k such that $j < k$. Now, by observation the maximum difference of 2 numbers among set of numbers will be formed by one number, being minimum. Example – let S = {2, 1, 3, 2} be the set of elements. Here the pair {1, 3} at index 1 and 2 respectively forms the maximum difference where 1 is the minimum until index 3. Based on this idea, we design an algorithm that can work on the input array A with n elements denoting price predictions for n days. We maintain variables for minimum price value till an index and the maximum profit by calculating the difference between the current price and minimum of values before that index. If this difference is more than current maximumProfit, we update the maximumProfit, and if the current stock price is lower than the minimum, we update the minimumPrice variable.

The day we buy the stock is maintained by the index where we found the minimum stock price and the day we sell the stock is maintained by the index where we found the maximum profit. 1 is added to these indices as the array indices start at 0.

In this algorithm for a single stock, we are iterating through set of n elements and performing n operations at maximum, thus time complexity for above is O(n).

We expand the logic above for a single stock to multiple stocks. This way, we will now have multiple 1d arrays, each denoting set of elements which are price predictions for different stocks. We just track the minimum stock price for each and compare the maximum profit obtained for each of stock $m_{i.}$

We run the n operations on array *m* times, thus the total time complexity to find single transaction with maximum profit, given an m*n matrix is *O(m x n)*.

## 2.2. Algorithm Design:

1. Use variable to store maximumProfit till any given transaction and initiate it to 0 and minimumPrice to save the minimum price of any given stock

2. Loop over each stock (1 to m)

3. For each stock, we calculate profit only if current price is greater than the minimumPrice of that stock

4. If profit of this Transaction is greater than the maximumProfit untill now, we update the maximumProfit value and the transaction details.

5. If current price is even less than the minimumPrice of that stock, we update the minimumPrice and we can buy a stock on this day to be sold in future.

6. Return the transaction details.

### Pseudo Code:

```
Maximum_Profit(m*n Array A):
        INITIALIZE maximumProfit = -1
        INITIALIZE minimumPrice = Large Integer Value
        INITIALIZE Stock Number, Stock buying day, Stock selling day = 0
        INITIALIZE nextPotentialBuyDay = 0
        FOREACH Stock i = 1 to m:
        FOREACH day j = 1, 2…. n:
        IF (CurrentPrice < MinimumPrice)
                MinimumPrice = CurrentPrice
                PotentialBuyDay = j
        ELSE IF (A[i][j] - minimumPrice > maximumProfit)
                Update maximumProfit = A[i][j] - minimumPrice
                Save the details of current transaction
                Stock Number = i
                Stock buying day = j
                Stock selling day = PotentialBuyDay
        END FOR
      END FOR
      PRINT/RETURN Transaction Details
```

## 2.3. Proof of Correctness:

**For 1 stock and 2 days** –

When the program runs for day 1, it updates the minimumPrice to price of stock on day 1 and potentialBuyDay to 1. In the next iteration, it checks if the price on day 2 is less than the currentMinimum, it will just update the minimumPrice and return. In this case, we won’t have any profit. However, if the price on day 2 is higher than that on day 1, it will update the maximumProfit and the transaction details as buy day = the already saved potentialBuyDay, selling day as day 2.

**For 1 stock and n days –**

The algorithm always keeps a track of the minimum stock price until an index and updates minimumPrice only if it finds an even lesser stock price. The algorithm updates the maximum profit value at any index k only if the difference between the current stock price and minimum price is greater than the current maximum profit, thus always maintaining the details for a maximum profit in the given array.

Let O be an optimal solution, it will always point to the transaction with maximum profit until any given transaction. Our algorithm calculates the profit if stock is bought on the day when the price was minimum and sold today. The maximum profit can either be the previous maximum or can be the difference of current price and the minimum price until now. Thus, our algorithm always saves the maximum profit and its corresponding transaction.

If we extend the same over **m stocks for n days**,

The minimum price is set every time when we consider a new stock, but the maximumProfit is compared with those from the previous stocks. This makes sure that we have a maximumProfit transaction among each stock.

## 2.4. Complexity:

The Algorithm Description stated above, provides a justification for the complexities stated below

- Time complexity: O(m*n) = We are iterating through set of *n* elements *m* times
- Space complexity: O(1) = Because only few variables - maximumProfit, minimumPrice, stock,buyDay,sellDay,potentialBuyDay are used. We do not use any additional memory that depends on the input size.

==================================================================================

**Algorithm Design Task:**

***ALG 3*** - Design a Θ(m ∗n) time dynamic programming algorithm for solving Problem1

This Algorithm was implemented in two Tasks as 3A & 3B using Memorization and Bottom-up approach respectively.

**3.1 & 3.2 Algorithm Description and Design:**

I. **Task 3A –** Algorithm 3A uses memorization to improve on recursive relations. For each stock, we first set the minimum price p to the stock price value of day 1. Upon iterating over each day i, we check if the difference *d* of *price[i] – p* is greater than the previously attained difference. This gives us the profit of transaction and index of sell operation. Otherwise, reset the minimum price to current index i. This can also be a new index at which stock is bought. The recurrence relation is defined as –

   A. if reached the end of row

      then return maximum profit obtained so far.

   B. If  d is greater than profit so far,

         D is new profit; update sell date and buy date.

   C. If current value at i is less than minimumPrice

         price[i] is the new minimumPrice, i can also be the new buy date.

   D. Do a, b, c on next index.

```
DEFINE N ← number of days,
        T ← contains buy date, sell date
        p ← minimum stock price
        i ← current day
        price ← array containing prices for stock m.
```

```
FindMaxProfit(price, i, p, profit, T):
    If i has reached N:
        Return profit
    Let d <- price[i] – p
    If d > profit:
        profit ← d,
        sell date is i,
    else if price[i] < p:
            p ← price[i],
            price[i] is new potential buy date
Return FindMaxProfit(price, i + 1, p, profit, T)
```

We expand the above function for multiple stocks m. For each stock j = 1 up to M , we execute *FindMaxProfit* . After each operation on stock m, we update T, collection of buy date, sell date and stock only if new maximum profit is obtained.

```
FindMaxProfit2D():
    DEFINE, M ← number of stocks,
                N ← number of days,
                Price[m][n] ← price of stock m on day n, i.e., 2d array
    INITIALIZE, T ← 0,0,0 i.e. stock, buy date, sell date,
                TT ← same as T, but only in has the scope of function call,
                Profit ← 0
    Foreach m = 1 up to M:
            TT = FindMaxProfit(price[m], 1, 0, 0, TT).
            If TT.profit > profit:
                Set profit to TT.profit
                Update T with values in TT.
                Set T.stock to m
    EndFor
    Return T.
```

**3.3. Proof of Correctness:**

We prove the correctness of algorithm 3a using induction –

For 1 stock and 1 day, the resulting profit is 0.

For 2 stock and 1 day, the resulting profit 0.

For 1 stock, 2 days, the resulting profit is max(0, price[day2] – price[day2]).

Expanding the solution to 1 stock, n days we get the profit which is maximum since we are tracking minimum price to the end. Profit changes only when minimum price changes.

Since the solution for m stocks is simply the extension of single stock, all the logic from previous statement also applies to it. Additionally, we update the final profit and transaction dates only when we have found new higher profit in some stock m > 1.

### 3.4. Complexity:

**I. 3A –** Time complexity for both approach is O(m*n). We. Find maximum profit for n days and for each stock m in a single pass resulting in T(n). Since we are performing this single pass operation m time, resulting time complexity is O(m*n).

Space complexity is O(m*n) dominated by the input. We. Use constant space of size 4 to retain transaction details.

==========================================================================

### 3B.1 Algorithm Description:

We can also solve this problem using memorization. Let P be the m*n array which denotes the prices for m stocks in n days. We use an m*n memo table to store the maximum profit of ith stock on jth day. This matrix is populated considering the fact that on each day, we can either do a transaction if we are getting more profit or we do not do the transaction. Let the memo table be called Profit. We also store the minimum price of a stock in some variable and keep updating it as soon as we find an even smaller price of that stock. The DP formulation for populating this matrix:

Profit[i][j] =

| | |
|---|---|
| 0 | , if i=j=0 |
| Profit[i-1][n] | , if i>0, j=0 |
| profit[i][j-1] | , if P[i][j] < minimum price of stock i |
| P[i][j] – minimumPrice | , if profit[i][j-1] < P[i][j] – minimumPrice (currentProfit) |
| Profit[i][j-1] | , otherwise |

We update the transaction details whenever we get current profit is greater than the maximum profit. Profit[m][n] represents the maximum profit obtained by one single transaction of any stock.

We run the n operations on array *m* times, thus the total time complexity to find single transaction with maximum profit, given an m*n matrix is *O(m x n)*.

An m*n matrix is used to store the maximum profit of each transaction, the space complexity is O(m*n)

## 3B.2. Algorithm Design:

1. Create an m*n memo table ‘profit’ to save the maximum profit of sellting ith stock upto jth day.

2. Loop for stock = 1 to m

3. For each stock, initialize it’s minimum price to it’s price on first day, and maximum profit on 0th day to be 0 if first stock and to profit[i-1][n]   for all other stocks.

4. Loop over each day j = 1 to n for each stock

5. If it’s price is less than minimum stock price, update minimum price and set profit on i,j be same as on (j-1) day i.e profit[i][j-1].

If the current transaction gives more profit than profit[i][j-1] , update profit[i][j] to be current profit and save truncation details.

If none of the condition is satisfied and we do not do a transaction on jth day, update profit table value to profit on previous day.

6. Return Transaction details.

Pseudo Code:-

```
MaximumProfit(m*n Array  A):
DEFINE n => number of columns of A,
m => number of rows of A
DEFINE Profit[i][j] = m*n matrix
INITIALIZE profit[0][0] = 0;

FOREACH stock i = 1 up to m:
    MinimumPrice = A[i][0],    // Set minimum price of a stock to its price on first day
    FOREACH day j = 1 up to n:
        If A[i][j] < minimumPrice:
            minimumPrice = A[i][j]
            profit[i][j] = profit[i][j-1]
            probableBuyDay = j
        else if: profit[i][j-1] < A[i][j] - minimumPrice
            profit[i][j] => A[i][j] – minimumPrice
```

```
// Save transaction details
Stock = i
BuyDay = probableBuyDay
sellDay = j
else:
profit[i][j] = profit[i][j-1]
ENDFOR
ENDFOR

PRINT/RETURN Transaction Details
```

## 3B.3. Proof of Correctness:

For the trivial case of only **1 stock and 2 days** –

There is only 1 possible transaction – buy stock on first day and sell on second day.

As per our algorithm, if this transaction gives profit(>maximumProfit i.e 0), it will return these transaction details.

For **1 stock and n days** –

We calculate the profit for transaction for buying it on the day the stock price was minimum and selling it on ith day. The transaction details is updated only if the current transaction's profit is greater than the maximum profit of any previous transactions, represented by the memorization matrix precious column on the same row. The memorization matrix always store the maximum profit of $i^{th}$ stock upto $j^{th}$ day.

Let O be an optimal solution, it will always point to the transaction with maximum profit until any given transaction. In our solution, profit[i][j] stores maximum profit and updates the transaction details only if the new transaction gives more profit than the previous maximum profit. Thus, our solution is optimal as it always stores the transaction that gives maximum profit.

Extending above for **m stocks and n days** – We perform the same process for every stock, and thus the transaction details will be updated only if we get more profit for any transaction of some other stock. Thus, transaction details always store data of the stock which gave maximum profit.

**All the transactions for which the profit is calculated are valid** – Since the day the stock is sold is always greater than the day it was bought, the transactions are always valid.

**The algorithm terminates -** Since we loop only for m stocks and for each stock, we loop only for valid transactions up to $n^{th}$ day. Thus, the algorithm will always terminate as it has limited valid transactions and we calculate profit only of the valid transactions.

**3B.4. Complexity:**

The Algorithm Description stated above, provides a justification for the complexities stated below

- Time complexity: O(m*n) = We are iterating through set of *n* elements *m* times
- Space complexity: O(m*n) = Because only a matrix to store the maximum profit of transactions.

**Problem Statement 2:**

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.]

**Algorithm Design Task:**

***ALG 4***- Design a $\Theta(m * n^{2k})$ time brute force algorithm for solving Problem2

**4.1 Algorithm Description:**

Given M as number of stocks and n as number of days, we simply seek to consider all the possibilities up to transaction K. Each transaction consists of exactly one buy and one sell. However, one can buy any stock from 1 to M on a given day n. After buying, same stock can be sold on any day n + i, where i = 0 to N such that n + i is less than or equal to N. On transaction t, one can again buy any stock from 1 to m repeating the operation, hence we consider the solution for each stock and add it to t. The algorithm is meant to give maximum profit after performing K transactions.

**4.2 Algorithm Design:**

Since this is brute force, we simply consider all the possibilities. On each day i, we can either buy or sell the stock. We set the sell flag upon buying and vice versa. Based on the flag, algorithm either performs buy operation or the sell operation. Let us represent maxProfit P as the maximum profit after each transaction. When it comes to buying, we have up to m options. Hence, we try all the stocks and chose the one with maximum profit when added to P. Let O be the optimal solution up to transaction k. There are 2 subproblems – buy and sell.

Base case 1 –

Maximum profit is 0 when there is no transaction. Hence,

```
If k == 0,
        Return 0
```

Base case 2 –

Cannot buy on last day or cannot sell after last day, so any transaction involving such condition will have no profit value. Hence,

```
If i == N,
        Return 0
```

```
On buy day i,
        For m = 1 up to M
                R = MAX(R, result from buying stock m)
        R = MAX(R, result from skipping i and buying on i + 1)
On sell day i,
        R = MAX(result from selling on day i, result from not selling on day i)
```

No selling/buying on day i, simply means that we algorithm will attempt the same on day i+1. The above rationale gives us the basis for designing brute force.

```
DEFINE M ← number of stocks,
        N ← number of days,
        K ← number of allowed transactions,
        A[M][N] ← stock prices of stock m on day n i.e., m x n array
INITIALIZE day ß 1 i.e., day 1,
                Stock ← 1, representing stock 1,
                k ← K, number of transactions.
                type ← buy/sell, start with buying on day 1,
                buyIdx ← buying index on the current transaction.
FindMaxProfit(day, stock, k, type, buyIdx):
```

```
If k is 0:
    Return 0
If day > N:
    Return 0
INITIALIZE,  R ← -999
If buy:
    Foreach stock m from 1 to M:
        R = MAX(R, -A[m][day] + FindMaxProfit(day + 1, m, k, sell, day))
    EndFor
    R = MAX(R, FindMaxProfit(day + 1, 0, k, buy, 0) // value of m doesn't matter.
Else:
    R = MAX(A[m][i] +  FindMaxProfit(day, m, k - 1, buy, day),
                    FindMaxProfit(day + 1, 0, k, sell, day)).
EndIf
Return R
```

We were not able to expand solution to track the buy date and sell date for each transaction t which will result in optimal solution. We managed to understand that we can pass the buy date to selling operation and retain the sell date. But tracking the dates only for the optimal solution was challenging.

**4.3 Proof of Correctness:**

Let O be the optimal solution and T be the solution given by algorithm. T contains maximum profit, buy date, sell date for each stock.

Using exchange argument**,**

We will assume that O and T have the same solution up to day n for transactions up to k. Let P be the profit. For, T(n + 1), we will be adding value greater than 0 to the already obtained profit. If k + 1 gives value x, then the new profit is P + x. However, since O is the optimal solution, it will also produce x' such that P + x' is the maximum value up to transaction k + 1. Hence, the algorithm 4 gives best answer possible for k transactions.

**4.4 Complexity:**

The complexity of brute force is $m * n^{2K}$ since upon buying each of the *M* stocks on *n* days, we are also selling it on every remaining day. This forms a recursion tree, where each level will have k nodes. We will have overall $2^k$ nodes, hence the time complexity for n days is $n^{2k}$ and buying m stocks on each day gives us, $m * n^{2k}$.

The space complexity on the other hand will be O(3n) as we aim to store only the buy date, sell date and stock for each day.

==================================================================

**Algorithm Design Task:**

***ALG 5***- Design a Θ(m ∗n2∗k) time dynamic programming algorithm for solving Problem2

## 5.1 Algorithm Description:

We optimized the brute force solution by storing the results for each day up to transaction t. This way we can avoid re-computation of already calculated subproblems.

## 5.2 Algorithm Design:

Pseudo code –

```
DEFINE A ← m x n matrix containing prices of stock m on day n.
        K ← number of allowed transactions
findMaximumProfit(A, m, n, k)

    1. Declare 2 variables to store max values. Say, v1 and v2.
    2. Declare an array M[k+1 x n+1]. Similarly, declare an array 'table' [k+1 x n+1].
    3. For p=0 to n, initialize M[0][p] as an empty string and table[0][p] as 0 For q=0
       to k initialize and table[q][0] as 0 and M[q][0] as an empty string.
    4. For x = 1 to k,
           a. For j from 0 to m,
                  i. from j to n
    5. Declare a string t as an empty string representing transaction containing
       buyDate, sellDate and stock.
    6. Declare and initialize max as 0, value1 as table[x][j-1]
    7. For y = 0 to n, initialize v2 as stock[i][j]-stock[i][y]+table[x-1][y]
    8. If v2 is greater than max then If M[x-1][y] not equal to them initialize temp as
       M[x-1][y] + format(j, v, j);
    9. Else t ← empty string + format(i, v, j), Initialize maximumProfit to
       MAX(maximumProfit, v2).
```

10. *If v1 > maximumProfit, then initialize t as A[x][j-1] and max as v1 If maximumProfit < table[x][j], then initialize transaction with M[x][i]. Initialize table[x][j] as maximum between maximumProfit and table[x][j]*
11. *Initialize M[x][j] with t.*
12. *return M.*

### 5.3 Proof of Correctness:

### 5.4 Complexity:

- Time complexity: O(m * $n^2$ * k)
- Space complexity: O(1)

==============================================================

## Algorithm Design Task:

***ALG 6***- Design a Θ(m ∗n ∗k) time dynamic programming algorithm for solving Problem2

This Algorithm was implemented in two Tasks as 6A & 6B using Memorization and Bottom-up approach respectively.

**Task 6A:**

### 6A.1 Algorithm Description:

We create a k*n matrix to save the maximum profit between any stock given k transactions on nth day. We populate this matrix using recursion for (k-1) transactions and so on. We use another k*m matrix to store the maximum difference of k transactions for mth stock. Once we build these matrices, we iterate over these matrixes to find the transactions that gave the maximum profit for k'th transaction.

### 6A.2. Algorithm Design:

1. Create a k*n table to store 'profit' and a k*m matrix to store the maximum difference between prices of stocks.

2. Update lower matrix maximum difference with zero and upper triangular matrix with negative of that stock's price on 0th days.

3. Call the GetMaximumProfit method which recursively populates both of these matrices recursively.

4. For each stock 1 to m, maximumDifference of a stock for k transactions is calculated as the maximum of the maximumDifference a stock for k transactions or the result we get from recurvively calling this function for k-1 transactions – the stock price on nth day

5. We update the maxProfit to the maximum of current maxProfit or the sum of stock price on nth day and maximumDifference of stock price for kth transaction.

6. Update the profit matrix value for k transaction on nth day with this value.

7. Once these are built, we call the findTransactions method to check the transaction that gave the maximum profit.

## 6A.3. Proof of Correctness:

For the trivial case of only **1 stock and 2 days , and k =1** –

There is only 1 possible transaction, our algorithm will save calculate maximum profit of 1 transaction and save it to the profit table. And the findTransactions will always return this transaction only. Thus the algorithm is correct for the most trivial case.

For **m stock and n days, 1 transaction** –

The profit matrix by any stock will store the maximum profit in it's first row at nth day. We will never have a profit greater than this value for nth day. Similarly, the maximum difference will always be the maximum difference between the stock price and the maximum profit. When iterating to find the transactions, the sum will always give the given stock value and thus will give the correct transaction.

For **m stock and n days, k transactions** –

We build the solution on top of the above solution. We will never have a profit greater than this value.

## 6A.4. Complexity:

The Algorithm Description stated above, provides a justification for the complexities stated below

- Time complexity: O(m*n*k) = For each transaction, we loop for each day and calculate maximum from among all of the stocks thus running m*n*k times.
- Space complexity: O(k* max(n,m)) = Because only a matrix to store the maximum profit of transactions with n columns and another matrix to store maximum difference of all stocks with m columns.

**=================================================================**

**Task 6B:**

## 6B.1 Algorithm Description:

We tried to use a different approach for this method, wherein instead of saving transaction details as list of integers, we create an object for transaction and create stock, buyDay and sellDay as it's class variables. This makes it easier to store values and easier to interpret.

We create a memo table 'profit' as we used in algorithm 3b, but here instead of saving maximumProfit attained by selling i$^{th}$ stock on j$^{th}$ day, we save the maximum profit attained by selling any stock on $j^{th}$ day, and i represent the number of transactions. We use another 1 D matrix to store the maximum difference between one of the profits of some day and the minimum value of the stock from that day.

Once, we have built both of these matrices, we perform a depth search and iterate over these matrices to get the transactions which gave us the maximum profit. If the profit for nth column is same as profit in (n-1) th column, it means that the maximum profit did not come from this transaction, so we decrement the column index. Once we find such an n where the max profit is not same as that for previous day, we know that this index represents the day we sell the stock.

To find out when the stock was bought, we check where the difference between the given prices matrix A becomes same as the difference between the current maximum profit and the maximum profit at any other day. Once we find an index where the difference is same, we save the details of the transaction and add it to the list of transactions.

Once we find such a transaction, we do not need to search any further, so we break the loop and continue to find the next transaction. Once we find all k transaction, we return this list of transactions.

## 6B.2. Algorithm Design:

1. Create an k*n memo table 'profit' to save the maximum profit of selling any stock with maximum of i transactions upto jth day.

2. Create a 1 D matrix of length m to save maximum difference between maximum profit and minimum value of stock on ith day

3. Loop for transaction = 1 to k, for each number of transactions we loop for each day and for each stock.

4. For each stock, we find the maximum difference between maximum profit on any day and minimum price of stock and save it to the 1D matrix.

5. We iterate over this profit matrix and find the transactions that gave the maximum profit for k transaction represented by last cell of the matrix.

6. We start from last row and last column, and find the sell day where maximum profit is not same as the previous day profit.

7. Next we find the buy day by checking the difference between the profits on two days and the stock prices on those days for k transactions.

8. When we find such a transaction, add it to the result list we decrement the number of transaction and continue to find the next transaction.

9. Return Transaction details.

**Pseudo Code-**

```
MaximumProfit(m*n Array  A, int k):

DEFINE n => number of columns of A,
  m => number of rows of A

DEFINE Profit[i][j] = k*n matrix
DEFINE difference[i] =  1D matrix with 1 to m cells representing maximum difference
INITIALIZE profit[i][i] = 0 if i=0 or j=0
INITIALIZE difference[i] = Minimum Integer value

FOREACH transaction i = 1 up to k:
FOREACH day j = 1 up to n:
FOREACH stock l = 1 up to m:
   difference[l] = Maximum of (current difference value and  dirrerence between maximum profit on
any day and minimum rice of stock)
              maxProfit =  maximum of the current maxProfit or sum of maximum difference and the
stock value
ENDFOR
profit[i][j] = Maximum of maxProfit or the maximum profit till (j-1) th day
ENDFOR
ENDFOR

Call dfs to find transactions and return/print these transactions.

dfs(m*n Array  A, k*n array Profit ):
DEFINE result = List<Transactions>
DEFINE transaction = k
DEFINE days = n
while transaction > 0 and days > 0 {
       if current profit of k transactions on nth day == k transaction on (n-1)th day
          decrement n // to find which day the stock was actually sold

       int sellDay = n;

       if k and n greater than 0 {
```

```
   FOREACH day = n to 1
      boolean b = false;
      FOREACH stock = 1 to m {
         if (stock price on sellDay - stock price on current day == maximum profit of k transaction
on nth day - maximum profit of k transactions on current day) {
               Create Transaction object and add to result list
            b = true;                              // if transaction found, no need to search
another buy day of this transaction
            break;
      ENDFOR
      if we found the trnsaction:
         break;
   }
}

ENDWHILE
RETURN result

class TransactionData{
class variables: stock, buyDay, sellDay;
}
```

## 6B.3. Proof of Correctness:

For the trivial case of only **1 stock and 2 days , and k =1** –

There is only 1 possible transaction, our algorithm will save calculate maximum profit of 1 transaction and save it to the profit table. And return this transaction. Thus the algorithm is correct for the most trivial case.

For **m stock and n days, 1 transaction** –

We calculate the maximum profit of any stock till jth day and save it to the first row of the matrix.

Thus, fist row will always give the maximum profit of any stock on that day. Since difference stores the maximum difference, when finding the transaction the sum will be equal always and thus returning the correct stock and the buy day of that stock.

For **m stock and n days, k transactions** –

Extending above fork transactions – We perform the same process for k transactions and bruild on top of the above result. The transaction details will be updated only if we find the transaction that gave maximum profit as given in the memoization matrix. Thus, transaction details always store transaction details that gave maximum profit.

**6B.4. Complexity:**

The Algorithm Description stated above, provides a justification for the complexities stated below

- Time complexity: O(m*n*k) = For each transaction, we loop for each day and calculate maximum from among all of the stocks thus running m*n*k times.
- Space complexity: O(k*n) = Because only a matrix to store the maximum profit of transactions.

**=================================================================**

## Conclusion:

**We have worked as a group of three members on this project. Working on this project was a great learning opportunity as we were able to apply the theoretical knowledge that we gained from classroom sessions about design and Analysis of algorithms and get hands-on experience working on designing and implementing multiple algorithms and analysing the various aspects of Algorithms. We have understood the crucial role of Time and Space complexity in designing efficient Algorithms.**

### Learnings:

- How to design algorithm with efficient space & time complexity

- Testing for corner cases is crucial for any algorithm

- time and space complexity tradeoff is crucial

- reuse of same algorithm design to implement multiple method

### Challenges:

- brute force for task 4 was hard to implement considering so many factors, transactions – stock – days etc

- task 1 , task 2, task 3a, task 3b were moderate to easy level difficulty to implement. We did not face much difficulties

-Task 4 was difficult to implement with brute-force approach. We were able to fetch maximum possible profit with the expected time complexity but was difficult to implement in java so implemented in c++

- Task 5, Task6a,Task6b were little difficult but were implemented

**Observations:**

| n=10,k=3 | M= 2 | M=4 | M=6 | M=10 |
|---|---|---|---|---|
| 1 | 1739375 | 1664708 | 2338375 | 1073250 |
| 2 | 1213125 | 1228459 | 2539750 | 1180125 |
| 3a | 1873166 | 1695334 | 3329958 | 1758500 |
| 3b | 2381166 | 2085542 | 1346459 | 3339625 |
| 4 | | | | |
| 5 | 38942959 | 38124292 | 48457458 | 47173625 |
| 6a | 2765962 | 2175990 | 4645621 | 6047726 |
| 6b | 2368875 | 1675959 | 4055333 | 5574167 |

| | M=10,k=3 | N=5 | 10 | 20 | 30 | 100 |
|---|---|---|---|---|---|---|
| 1 | | 1527584 | 1282917 | 1443125 | 1724500 | 5781875 |
| 2 | | 2027709 | 1363292 | 2615041 | 1580584 | 1888792 |
| 3a | | 2448791 | 1853291 | 1048750 | 2265166 | 1231750 |
| 3b | | 1260042 | 2310041 | 2910083 | 2030833 | 1457084 |
| 4 | | | | | | |
| 5 | | 33650375 | 49997667 | 68177875 | 71577250 | 110224667 |
| 6a | | 4930493 | 9058406 | 6511234 | 92213201 | 6003732 |
| 6b | | 4750792 | 8658708 | 6265333 | 88319500 | 5603625 |

n=10,m=1

| 0 | k=2 | k=4 | k=6 | k=10 |
|---|---|---|---|---|
| 5 | 35699625 | 65687334 | 63128583 | 75141125 |
| 6a | 7621868 | 8712934 | 5027821 | 8923132 |
| 6b | 6511917 | 8920583 | 4718750 | 8772042 |

**Note:**

**All the three members of the project has given equal contribution towards design, implementation and documentation.**