COT5405 Algorithms Programming Project

1 Team Members

Rachana Gugale UFID: 6353-2454 Email: rgugale@ufl.edu
Simran Kukreja UFID: 7207-0369 Email: s.kukreja@ufl.edu
Shromana Kumar UFID: 7062-1103 Email: shromana.kumar@ufl.edu

We had a few discussions initially to decide on the approach and algorithm to follow for each of the 12 problems. After the initial brainstorming, we had taken up 4 programming implementations each. During code debugging, we have helped each other to resolve and rectify our respective solutions. Once the implementation was completed, we compiled each of our individual implementations together, and generated a make file. For the final report we had multiple sessions where we all collaborated and came up with the final version of the report.

2 Design and Analysis of Algorithms

2.1 Problem1

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

2.1.1 Brute Force - Algorithm

The algorithm iterates over each stock. For each stock, it considers all possible buy and sell days by running two nested loops. For each buy and sell day combination, it checks if the profit is greater than the maximum profit found till now. The stock index and indices of buy and sell days which give the maximum profit are maintained.

2.1.2 Brute Force - Proof of Correctness

Proof by Contradiction:

- Let's assume $maxProfit$ given by our algorithm does not actually contain the maximum profit.
- The output given by our algorithm is $(stockId, buyDay, sellDay)$.
- Since our algorithm doesn't give the optimal output, there has to be an optimal output $O(stockId', buyDay', sellDay')$ such that $profit' > maxProfit$.
- When iterating over $stockId'$ for $buyDay'$ and $sellDay'$, there would have come a case where $profit' > maxProfit$. Hence, $maxProfit$ would have been updated to $profit'$.
- But this contradicts our assumption that $profit' > maxProfit$. Hence our algorithm correctly returns the maximum profit.

Algorithm 1 Alg1: Design $O(m * n^2)$ time brute force algorithm for solving Problem1

```
function BRUTEFORCE(m, n, priceMatrix)

    maxProfit ← −∞
    maxProfitStockIdx ← −1
    maxBuyIdx ← −1
    maxSellIdx ← −1

    for stockIdx = 0 to m do
        for buyDayIdx = 0 to n − 1 do
            for sellDayIdx = buyDayIdx + 1 to n do
                currDiff ← priceMatrix[stockIdx][sellDayIdx] − priceMatrix[stockIdx][buyDayIdx]

                if currDiff > maxProfit then
                    maxProfit ← currDiff
                    maxProfitStockIdx ← stockIdx
                    maxBuyIdx ← buyDayIdx
                    maxSellIdx ← sellDayIdx
                end if
            end for
        end for
    end for
    return {maxProfitStockIdx + 1, maxBuyIdx + 1, maxSellIdx + 1}
                                                        ▷ +1 for 1-based indexing
end function
```

2.1.3 Brute Force - Time and Space Complexity Analysis

Time Complexity: As there are three loops (one iterating over stocks and two iterating over days) and the work done by each iteration of the innermost loop is constant, the time complexity of the algorithm is $O(m * n^2)$

Space Complexity: Only constant number of integer variables are used by the algorithm to store the maximum profit, the stock, buy and sell indices associated with it. So the space complexity is $O(1)$.

2.1.4 Greedy - Algorithm

In the greedy approach, stocks are bought at the minimum price found so far and sold at the maximum price found so far. We maintain indices of the minimum and maximum stock price found so far for each stock. For any given day, we buy the stock at the minimum price available till that day, and measure how much maximum profit (if any) we will be able to get if we sell the stocks on the current day. We keep recalculating the maximum profit for every day for each stock. This way we would have found the maximum profit for each of the m different stocks. After that, we choose the stocks with the maximum profit.

2.1.5 Greedy - Proof of Correctness

- **Assertion P(i):** $maxProfit$ stores the maximum profit obtained from i stocks, considering i stocks and their prices on n days.
- **Base case:** Base case: For $m = 1$, we are returning the $maxProfit$ calculated using $(maxProfit < currProfit)$, for all the n days, comparing 2 days at a time, and as $maxProfit$ contains the maximum profit value for that stock, thus the assertion holds.
- **Inductive Hypothesis:** Let's assume that the algorithm returns the accurate $maxProfit$ for the $P(i-1)$ stock, thus, the assertion on $P(i)$ would also hold.
- **Inductive Step:** To compute the value of $P(m)$, we iterate over all the stocks i.e., $1 < stock <= m$, and then iterate over all the days from $1 <= day <= n$.
  This results in calculating $maxProfit$ for each day using $maxProfit = maxmaxProfit, currProfit$ for day-based profit comparisons and $finalProfit < maxProfit$ for stock-based profit comparisons.
  For a day, if $maxProfit < currProfit$, we update $maxStockIndex = dayIdx$, and if $minStockValue < currProfit$, we update $minStockIndex = minStockCounterIndex$.
  Hence, we receive the maxProfit at the end of the n(th) day iteration through getting minimum stock values for buy days and maximum profit on sell days.
  For the m(th) stock iteration, we get the maxProfit by comparison of finalProfit i.e., $P(i-1) < maxProfit$, which again returns the maximum profit for every stock. Hence, the final value returned contains the stock and buy, sell day indices that return maxProfit.

2.1.6 Greedy - Time and Space Complexity Analysis

Time Complexity: As there are two loops (one iterating over stocks and one iterating over days) and the work done by each iteration of the innermost loop is constant, the time complexity of the algorithm is $O(m * n)$.

Space Complexity: Constant number of integer variables are used by the algorithm to store the maximum profit, the stock, buy and sell indices and various intermediate values associated with it. So the space complexity is $O(1)$.

Algorithm 2 Alg2: Design a $O(m*n)$ time greedy algorithm for solving Problem1

```
function GREEDY(m, n, priceMatrix)

    finalStockIndex ← 1
    finalBuyPrice ← priceMatrix[0][0]
    finalSellPrice ← priceMatrix[0][0]
    finalBuyIndex ← 0
    finalSellIndex ← 0
    finalProfit ← −∞

    for stockIdx = 0 to m do
        maxProfit ← −∞
        minStockCounter ← priceMatrix[stockIdx][0]
        minStockCounterIndex ← 0
        maxStockValue ← priceMatrix[stockIdx][0]
        minStockValue ← priceMatrix[stockIdx][0]
        maxStockIndex ← 0
        minStockIndex ← 0

        for dayIdx = 0 to n do
            currProfit ← priceMatrix[stockIdx][dayIdx] − minStockCounter
            if maxProfit < currProfit then
                maxStockValue ← priceMatrix[stockIdx][dayIdx]
                maxStockIndex ← dayIdx
            end if

            if minStockValue < currProfit then
                minStockValue ← minStockCounter
                minStockIndex ← minStockCounterIndex
            end if

            maxProfit ← max{maxProfit, currProfit}
            if minStockCounter >= priceMatrix[stockIdx][dayIdx] then
                minStockCounterIndex ← dayIdx
            end if

            minStockCounter ← min{minStockCounter, priceMatrix[stockIdx][dayIdx]}
        end for

        if finalProfit < maxProfit then
            finalSellIndex ← maxStockIndex + 1
            finalBuyIndex ← minStockIndex + 1
            finalSellPrice ← maxStockValue
            finalBuyPrice ← minStockValue
            finalStockIndex ← stockIdx + 1
        end if
        finalProfit ← max{finalProfit, maxProfit}
    end for
    return {finalStockIndex, finalBuyIndex, finalSellIndex}

end function
```

2.1.7 Dynamic Programming - Algorithm

The dynamic programming algorithm calculates and stores the maximum profit transaction for each stock in an array $maxPerStock$. The final solution then is the maximum profit transaction in $maxPerStock$.

Recursive Formulation:

$$OPT(i,j) = \begin{cases} 0 & \text{for OPT(i, 0)} \\ max\{0, OPT(i,j-1) + (price[i][j] - price[i][j-1])\} & \text{Otherwise} \end{cases}$$

$OPT(i,j)$ denotes the maximum possible profit obtained by trading stock i till day j.

The recursive function has 2 cases:

Case1: $Profit = 0$
In this case, we either hold a stock we have already bought, buy a new stock or do nothing. In all these scenarios, the profit is 0.

Case2: $Profit = OPT(i,j-1) + (price[i][j] - price[i][j-1])$
In this case, we sell the stock we are holding and obtain a profit.

Since we want to maximize our profit, we take the maximum of both cases.

2.1.8 Dynamic Programming - Proof of Correctness

- **Assertion P(i):** maxProfit stores the maximum profit obtained from i stocks, considering i stocks and their prices on n days.
- **Base case:** For m=1, we are returning the $maxPerStock[stockIdx]$ calculated using $(OPT[stockIdx][dayIdx] > maxPerStock[stockIdx])$, for all the n days, comparing 2 days at a time, and as $maxPerStock[stockIdx]$ contains the maximum profit value for that stock, thus the assertion holds.
- **Inductive Hypothesis:** Let's assume that the algorithm returns the accurate $maxPerStock[stockIdx]$ for the P(i-1) stock, thus, the assertion on P(i) would also hold.
- **Inductive Step:** To compute the value of P(m), we iterate over all the stocks i.e., $1 < stock <= m$, and then iterate over all the days from $1 <= day <= n$. This results in calculating maxPerStock[stockIdx] for each day by using $OPT[stockIdx][dayIdx] > maxPerStock[stockIdx]$ for day-based profit comparisons and $maxProfit$ ¸ $max(maxPerStock)$ for stock-based profit comparisons.
  For a day, if $OPT[stockIdx][dayIdx] > maxPerStock[stockIdx]$, we update $maxPerStock[stockIdx]$ ¸ $OPT[stockIdx][dayIdx]$, and if $OPT[stockIdx][dayIdx]$ = 0, we update $maxBuyIndices[stockIdx]$ ¸ $dayIdx$.
  Hence, we receive the maxPerStock[stockIdx] at the end of the n(th) day iteration along with the buying and selling indices.
  For the m(th) stock iteration, we get the maxProfit by comparison of maxPerStock i.e., P(i-1) with maxProfit, which returns the maximum profit out of all the stocks.
  Hence, the final value returned contains the stock and buy, sell day indices that return maxProfit.

Algorithm 3 Alg3: Design $O(m * n)$ time dynamic programming algorithm for solving Problem1

function DP($m, n, priceMatrix$)

OPT is 2D array of size m*n

for $i = 0$ *to* m **do**

$OPT[i][0] = 0$

end for

$maxPerStock \leftarrow$ 1D array of size m with all elements initialized to -∞

$maxBuyIndices$ is 1D array of size m

$maxSellIndices$ is 1D array of size m

for $stockIdx = 0$ *to* m **do**

for $dayIdx = 1$ *to* n **do**

$OPT[stockIdx][dayIdx] \leftarrow max\{0, OPT[stockIdx][dayIdx-1] + (priceMatrix[stockIdx][dayIdx] - priceMatrix[stockIdx][dayIdx-1])\}$

if $OPT[stockIdx][dayIdx] = 0$ **then**

$maxBuyIndices[stockIdx] \leftarrow dayIdx$

end if

if $OPT[stockIdx][dayIdx] > maxPerStock[stockIdx]$ **then**

$maxPerStock[stockIdx] \leftarrow OPT[stockIdx][dayIdx]$

$maxSellIndices[stockIdx] = dayIdx$

end if

end for

end for

$maxProfit \leftarrow max(maxPerStock)$

$maxProfitIndex \leftarrow$ Stock which gives maxProfit

return $\{maxProfitIndex + 1, maxBuyIndices[maxProfitIndex] + 1, maxSellIndices[maxProfitIndex] + 1\}$

$\triangleright$ *+1 for 1-based indexing*

end function

2.1.9 Dynamic Programming - Time and Space Complexity Analysis

Time Complexity: There are two loops, the outer loop iterating on the number of stocks from 1 to m having a time complexity of $O(m)$, and the inner loop iterating on days, from 1 to n, having a time complexity of $O(n)$. Hence, the time complexity of the combined nested loops is $O(m * n)$.

Space Complexity: The algorithm is using three 1D arrays to store the stock, buy day, and sell day indices, each of length m. Hence, the complete space complexity of the algorithm is $O(m)$

2.2 Problem2

Problem Definition: 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), and a sequence of at most k transactions that gives maximum profit.

2.2.1 Brute Force - Algorithm

2.2.2 Brute Force - Proof of Correctness

- **Assertion P(k):** maxProfit stores the maximum profit at the end of at most k transactions, considering m stocks and their prices on n days.
- **Base case:** For k=1, we get the maximumProfit obtained by iterating over all the stocks ranging from $1 <= stock <= m$ and the days i.e., $1 <= day <= n$, considering all the stocks for the buy day, and sell day, and picking the stock where profit is maximum.
- **Inductive Hypothesis:** Let's assume that the algorithm returns the accurate maxProfit for the $P(k-1)$ transaction, thus, the assertion on P(k) would also hold.
- **Inductive Step:** To compute the value of P(k), we recursively call the function for k transactions, going from k to 0 occurring on n days, ranging from 0 to n-1.
  Each recursive call checks if the current scenario is buying or selling a share.
  In the buy scenario, if we're holding the stock for the current day, then profit is recursively calculated for buying the stock on the next day:
  $tmpMaxProfit, tmpTxnList = bruteForce(stockIdx, buyIdx, dayIdx{+}1, currTxn, canSell, txnList)$.
  If we are selling the stock, then profit is calculated using
  $diff = priceMatrix[stockIdx][dayIdx]priceMatrix[stockIdx][buyIdx]$ and recursively computing the buy value for the next transaction
  $tmpMaxProfit, tmpTxnList = bruteForce(null, null, dayIdx, currTxn1, false, modifiedTxnList)$.
  In the selling scenario, we either skip the day for selling and recursively compute selling on the next day
  $tmpMaxProfit, tmpTxnList = bruteForce(null, null, dayIdx + 1, currTxn, false, txnList)$ or sell on that day and compute buying maxProfit recursively for the next day:
  $tmpMaxProfit, tmpTxnList = bruteForce(i, dayIdx, dayIdx + 1, currTxn, true, txnList)$.
  For each of these recursive calls, we are comparing the current profit with the maxProfit: (currProfit i.e., P(i-1) $>$ maxProfit) and the final value returned is hence a transaction list holding the maxProfit buy and sell days for at most k transactions.

2.2.3 Brute Force - Time and Space Complexity Analysis

Time Complexity: In this algorithm, each day is considered for buy and sell, and hence we're recursively computing the profit, comparing the stock prices of two days at a time, resulting in a time complexity of $O(n^2)$ for the complete day comparison. Further, for every buy scenario, we are considering all the stocks from 1 to m for buying, resulting in $O(m * n^2)$ complexity. This process is being done for each transaction, and hence, the total time complexity is $m * (n^2)^k$, since we are computing the values of one transaction and then having to re-compute the values for the previous transaction.

Space Complexity: The algorithm is using a list to store the stock, buy day, and sell day indices. Since at maximum there would be k entries in this list which is equivalent to the number of transactions. Hence, the complete space complexity of the algorithm is $O(k)$.

Algorithm 4 Alg4: Design a $O(m * n^(2k))$ time brute force algorithm for solving Problem2

```
function BRUTEFORCE(stockIdx, buyIdx, dayIdx, currTxn, canSell, txnList)

    maxProfit ← 0
    finalTxnList ← []

    if (dayIdx >= priceMatrix[0].length)||(currTxn == 0) then
        return {maxProfit, txnList}
    end if

    ▷ We have a stock ◁
    if (canSell == true) then

        ▷ Holding the stock ◁
        {tmpMaxProfit, tmpTxnList} ← bruteForce(stockIdx, buyIdx, dayIdx +
        1, currTxn, canSell, txnList)
        currProfit ← tmpMaxProfit
        currTxnList ← tmpTxnList

        if (currProfit > maxProfit) then
            maxProfit ← currProfit
            finalTxnList ← currTxnList
        end if

        ▷ Selling the stock ◁
        diff ← priceMatrix[stockIdx][dayIdx] − priceMatrix[stockIdx][buyIdx]
        modifiedTxnList ← txnList
        modifiedTxnList.add([stockIdx, buyIdx, dayIdx])

        {tmpMaxProfit, tmpTxnList} ← bruteForce(null, null, dayIdx, currTxn −
        1, false, modifiedTxnList)

        currProfit ← tmpMaxProfit
        currTxnList ← tmpTxnList
        currProfit ← currProfit + diff

        if (currProfit > maxProfit) then
            maxProfit ← currProfit
            finalTxnList ← currTxnList
        end if

    else                                                        ▷ We don't have a stock
        ▷ Skip the day ◁
        {tmpMaxProfit, tmpTxnList} ← bruteForce(null, null, dayIdx + 1, currTxn, false, txnList)
        currProfit ← tmpMaxProfit
        currTxnList ← tmpTxnList

        if (currProfit > maxProfit) then
            maxProfit ← currProfit
            finalTxnList ← currTxnList
        end if
```

```
        ▷ Buy stock of one of the companies ◁
        for i = 0 to priceMatrix.length do
            {tmpMaxProfit, tmpTxnList} ← bruteForce(i, dayIdx, dayIdx + 1, currTxn, true, txnList)
            currProfit ← tmpMaxProfit
            currTxnList ← tmpTxnList

            if (currProfit > maxProfit) then
                maxProfit ← currProfit
                finalTxnList ← currTxnList
            end if

        end for

    end if

    return {maxProfit, finalTxnList}
end function
```

2.2.4 Dynamic Programming $O(m * n^2 * k)$ - Algorithm

Recursive Formulation:

OPT(t, i) = (0 for OPT(t, 0)

0 for OPT(0, i)

max{OPT(t, i -1), max{max{price[s][i] – price[s][j]} + OPT(t-1, j) }}

for all s and for all j in range [0, i-1] Otherwise

OPT(t, i) denotes the maximum possible profit obtained by selling on day i for txn t.

The recursive function has 3 cases:

Case 1: Profit = 0
We do not make any profit when txn t = 0 or when day i = 0

Case 2: Profit = OPT(t, i -1)
OPT(t, i) does not select day i. So, there must be an optimal solution for days 1 to (i-1)

Case 3: Profit = max{max{price[s][i] – price[s][j]} + OPT(t-1, j) }
1. We first calculate the max{price[s][i] – price[s][j]} for all values of s
2. We select the best profit from the previous transaction OPT(t-1, j) to the maximize the profit calculated in step 1 for all j in range [0, i-1]

Figure 1: Recurrence Relation

2.2.5 Dynamic Programming $O(m * n^2 * k)$ - Proof of Correctness

We would prove the correctness by induction –

- **Assertion** – At the end of kth (txnloop), nth (dayloop) iteration, the $profit[txnloop][dayloop]$ would hold the maximum profit.
- **Base Case** – When txnloop = 1 and dayloop = 1, the $profit[txnloop][dayloop]$ would correctly return the maximum profit by buying at day0 and selling at day1 for the stock which returns the max profit.
- **Inductive Hypothesis** – Let's assume that algorithm gives maximum profit correctly for $profit[txnloop][dayloop–1]$ for the txnloopth and (dayloop – 1)th iteration, we would try to prove that it provides the correct output for $profit[txnloop][dayloop]$ for the txnloopth and dayloopth iteration.
- **Inductive Step –**
  • We calculate the maximum profit that can be made on the last day (dayloop = n) for all the previous buy days and stocks. We perform summation of previous profit value at $profit[txnloop - 1][buyloop]$ and the max profit that can be made at dayloop $max(priceMatrix[stockloop][dayloop] - priceMatrix[stockloop][buyloop])$ where buyloop is (0 to dayloop -1) and stockloop is (0 to total number of stocks). We store this maximum profit for the current iteration of dayloop (last day) in $maxTotalCurrProfit$.
  • We calculate $profit[txnloop][dayloop] = max(profit[txnloop][dayloop - 1], maxTotalCurrProfit)$ which takes the maximum profit between the profit calculated in $txnloop - th$ and $(dayloop–1)th$ iteration and the txnloop-th and dayloop-th iteration.
  • Thus, our assertion holds true that the maximum profit is returned.

Algorithm 5 Alg5: Design a $O(m * n^2 * k)$ time dynamic programming algorithm for solving Problem2

```
function DPN2(priceMatrix, noOfDays, noOfStocks, k)

    profit ← 2D array of size (k + 1) * noOfDays
    stock ← 2D array of size (k + 1) * noOfDays
    buy ← 2D array of size (k + 1) * noOfDays

    for txnloop = 0 to k do
        profit[txnloop][0] ← 0
        stock[txnloop][0] ← 0
        buy[txnloop][0] ← 0
    end for

    for dayloop = 0 to noOfDays do
        profit[0][dayloop] ← 0
        stock[0][dayloop] ← 0
        buy[0][dayloop] ← 0
    end for

    for txnloop = 1 to k do
        for dayloop = 1 to noOfDays do

            maxTotalCurrProfit ← 0
            buyIndex ← Integer.MINVALUE
            buyStockIndex ← 0

            for buyloop = 0 to dayloop do
                prevProfit ← profit[txnloop − 1][buyloop]
                maxcurrProfit ← 0
                stockIndex ← 0

                for stockloop = 0 to noOfStocks do
                    currProfit ← priceMatrix[stockloop][dayloop] − priceMatrix[stockloop][buyloop]
                    if currProfit >= maxcurrProfit then
                        stockIndex ← stockloop
                    end if

                    maxcurrProfit ← max(currProfit, maxcurrProfit)
                end for

                profitHere ← maxcurrProfit + prevProfit

                if (maxTotalCurrProfit >= profitHere) then
                    buyIndex ← buyloop
                end if

                if (maxTotalCurrProfit < profitHere) then
                    buyIndex ← buyloop
                    buyStockIndex ← stockIndex
                end if

                maxTotalCurrProfit = max(maxTotalCurrProfit, profitHere)
```

```
                if (profit[txnloop][dayloop − 1] < maxTotalCurrProfit) then
                    stock[txnloop][dayloop] ← buyStockIndex + 1
                    buy[txnloop][dayloop] ← buyIndex
                else
                    stock[txnloop][dayloop] ← stock[txnloop][dayloop − 1]
                    buy[txnloop][dayloop] ← buy[txnloop][dayloop − 1]
                end if

                profit[txnloop][dayloop] ← max(profit[txnloop][dayloop − 1], maxTotalCurrProfit)
            end for
        end for
    end for

    finalAns ← 2D array of size k*3

    for txnloop = k to 1and dayloop = (noOfDays − 1) to 1 do
        if (profit[txnloop][dayloop] == profit[txnloop][dayloop − 1]) then
            txnloop + +
            continue
        end if

        finalAns[txnloop − 1][0] ← stock[txnloop][dayloop]
        finalAns[txnloop − 1][1] ← buy[txnloop][dayloop] + 1
        finalAns[txnloop − 1][2] ← dayloop + 1

        dayloop ← buy[txnloop][dayloop] + 1
    end for

    return finalAns
end function
```

2.2.6 Dynamic Programming $O(m * n^2 * k)$ - Time and Space Complexity Analysis

Time Complexity: There are four loops, the outermost loop iterating on the number of transactions, from 1 to k having a complexity $O(k)$, and the next inner loop iterating on the days, ranging from 1 to n-1 having a complexity $O(n)$. The next loop is a loop on the buy days going from 0 to the day in the second loop having a worst-case complexity of $O(n)$. The last loop is iterating on the number of stocks from 1 to m having a complexity $O(m)$. Hence, the overall time complexity of the four nested loops is $O(m * n^2 * k)$

Space Complexity: The algorithm is using three 2D arrays to store the stock, buy day, and sell day indices, each of size (n*k). Hence, the complete space complexity of the algorithm is $O(n * k)$

2.2.7 Dynamic Programming $O(m * n * k)$ - Algorithm

Recursive Formulation:

OPT(t, i) = (0 for OPT(t, 0)
0 for OPT(0, i)
max{OPT(t, i -1), max{ price[s][i], max{OPT(t-1, j) – price[s][j] } } }

for all s and for all j in range [0, i-1] Otherwise

OPT(t, i) denotes the maximum possible profit obtained by selling on day i for transaction t.

The recursive function has 3 cases:

Case 1: Profit = 0
We do not make any profit when transaction t = 0 or when day i = 0

Case 2: Profit = OPT(t, i -1)
OPT(t, i) does not select day i. So, there must be an optimal solution for days 1 to (i-1)

Case 3: Profit = max{ price[s][i], max{OPT(t-1, j) – price[s][j] } }

1. Take the price[s][i] for all values of s
2. We are recursively executing OPT(t-1, j) for all j in range [0, i-1]. We take the maximum of {OPT(t-1, j) – price[s][j]}
3. Taking the maximum value returned from step 2, we would add the price from step 1 to it and check for all stocks to get the maximum profit

Figure 2: Recurrence Relation

2.2.8 Dynamic Programming $O(m * n * k)$ - Proof of Correctness

We would prove the correctness by induction –

- **Assertion** – At the end of kth (txnloop), nth (dayloop) iteration, the $profit[txnloop][dayloop]$ would hold the maximum profit.
- **Base Case** – When txnloop = 1 and dayloop = 1, the $profit[txnloop][dayloop]$ would correctly return the maximum profit by buying at day0 and selling at day1 for the stock which returns the max profit.
- **Inductive Hypothesis** – Let's assume that algorithm gives maximum profit correctly for $profit[txnloop][dayloop–1]$ for the txnloop-th and (dayloop – 1)th iteration, we would try to prove that it provides the correct output for $profit[txnloop][dayloop]$ for the txnloop-th and dayloop-th iteration.
- **Inductive Step** –
  • We have already calculated the maximum previous difference until (dayloop - 2) where previous difference is defined as $max(profit[txnloop - 1][j]–priceMatrix[j])$ where j is (0 to dayloop -2).
  • We calculate the previous difference for (dayloop -1) and store it in the variable $yesterdaysDiffwithProfit$.
  We calculate $prevDiffWithProfit[stockloop] \leftarrow max(prevDiffWithProfit[stockloop], yesterdaysDiffwithProfit)$ which stores the maximum previous difference until (dayloop - 1) and stockloop is (0 to total number of stocks).
  • We calculate the maximum profit that we get by comparing the $profit[txnloop][dayloop–1]$ and the $profit[txnloop][dayloop][stockloop]$ where stockloop is (0 to total number of stocks). We store the maximum returned in $profit[txnloop][dayloop]$.
  • Thus, the profit for $profit[txnloop][dayloop]$ takes the maximum profit between the profit calculated in txnloop-th and (dayloop – 1)th iteration and the txnloop-th and dayloop-th iteration.
  • Thus, our assertion holds true that the maximum profit is returned.

Algorithm 6 Alg6: Design a O(m * n * k) time dynamic programming algorithm for solving Problem2

```
function DPN(priceMatrix, noOfDays, noOfStocks, k)

    profit ← 2D array of size (k + 1) * noOfDays
    stock ← 2D array of size (k + 1) * noOfDays
    buy ← 2D array of size (k + 1) * noOfDays
    prevDiffWithProfit ← 1D array of size noOfStocks
    buyIndex ← 1D array of size noOfStocks

    for txnloop = 0 to k do
        profit[txnloop][0] ← 0
        stock[txnloop][0] ← 0
        buy[txnloop][0] ← 0
    end for

    for dayloop = 0 to noOfDays do
        profit[0][dayloop] ← 0
        stock[0][dayloop] ← 0
        buy[0][dayloop] ← 0
    end for

    for txnloop = 1 to k do
        for stockloop = 0 to noOfStocks do
            prevDiffWithProfit[stockloop] ← Integer.MINVALUE
            buyIndex[stockloop] ← 0
        end for

        for dayloop = 1 to noOfDays do
            yesterdaysProfit ← profit[txnloop − 1][dayloop − 1]
            for stockloop = 0 to noOfStocks do

                yesterdaysDiffwithProfit ← yesterdaysProfit − priceMatrix[stockloop][dayloop − 1]

                if prevDiffWithProfit[stockloop] <= yesterdaysDiffwithProfit then
                    buyIndex[stockloop] ← dayloop − 1
                end if

                prevDiffWithProfit[stockloop] ← max(prevDiffWithProfit[stockloop], yesterdaysDiffwithProfit)

                tempMax ← max(profit[txnloop][dayloop − 1], priceMatrix[stockloop][dayloop] +
                prevDiffWithProfit[stockloop])

                if tempMax > profit[txnloop][dayloop] then
                    stock[txnloop][dayloop] ← stockloop + 1
                    buy[txnloop][dayloop] ← buyIndex[stockloop]
                end if

                profit[txnloop][dayloop] ← max(tempMax, profit[txnloop][dayloop])
            end for

        end for

    end for
```

```
    finalAns ← 2D array of size k*3

    for txnloop = k to 1 and dayloop = (noOfDays − 1) to 1 do
        if (profit[txnloop][dayloop] == profit[txnloop][dayloop − 1]) then
            txnloop++
            continue
        end if

        finalAns[txnloop − 1][0] ← stock[txnloop][dayloop]
        finalAns[txnloop − 1][1] ← buy[txnloop][dayloop] + 1
        finalAns[txnloop − 1][2] ← dayloop + 1

        dayloop ← buy[txnloop][dayloop] + 1
    end for

    return finalAns
end function
```

2.2.9 Dynamic Programming $O(m * n * k)$ - Time and Space Complexity Analysis

Time Complexity: There are three loops, the outermost loop iterating on the number of transactions, from 1 to k having a complexity O(k), and the next inner loop iterating on the days, ranging from 1 to n-1 having a complexity O(n). The third loop is iterating on the number of stocks from 1 to m having a complexity O(m). Hence, the overall time complexity of the three nested loops is $O(m * n * k)$

Space Complexity: The algorithm is using three 2D arrays to store the stock, buy, and sell values, each of size (n*k). Additionally, we are using two 1D arrays of size m to store buy indices and profit differences. Hence, the complete space complexity of the algorithm is $O(n * k)$

2.3 Problem3

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

2.3.1 Brute Force - Algorithm

2.3.2 Brute Force - Proof of Correctness

- **Assertion P(i):** maxProfit stores the maximum profit obtained at the end of i days with the cooldown period, considering m stocks and their prices on n days.
- **Base case:** For n=1, we get the maximumProfit by iterating over all the stocks ranging from 1¡=stock¡=m on day 1, considering all the stocks for buying on day 0 and selling on day 1, and picking the stock where profit is maximum.
- **Inductive Hypothesis:** Let's assume that the algorithm returns the accurate maxProfit for the P(i-1) transaction, thus, the assertion on P(i) would also hold.
- **Inductive Step:** To compute the value of P(i), we recursively call the function for I days, ranging from 0 to n-1. Each recursive call checks if the current scenario is buying or selling a share. In the buy scenario, if we're holding the stock for the current day, then profit is recursively calculated for buying the stock on the next day: tmpMaxProfit, tmpTxnList = bruteForce(stockIdx, buyIdx, dayIdx + 1, currTxn, canSell, txnList).
  If we are selling the stock, then profit is calculated using diff = priceMatrix[stockIdx][dayIdx] - priceMatrix[stockIdx][buyIdx] and recursively computing the buy value for the next transaction tmpMaxProf it, tmpTxnList = bruteForce(null, null, dayIdx, currTxn 1, false, modifiedTxnList).
  The sell scenario begins after the cooldown period: In the selling scenario, we are either skipping the day for selling and recursively computing profit on selling on the next day tmpMaxProfit, tmpTxnList = bruteForce(null, null, dayIdx + 1, currTxn, false, txnList) or selling on that day and computing buying maxProfit recursively on the next day: tmpMaxProf it, tmpTxnList = bruteForce(i, dayIdx, dayIdx+ 1, currT xn, true, txnList). For each of these recursive calls, we are comparing the current profit with the maxProfit: (currProfit i.e. P(i-1) ¿ maxProfit) and the final value returned is hence a transaction list holding the maxProfit buy and sell days for at most k transactions.

2.3.3 Brute Force - Time and Space Complexity Analysis

Time Complexity: In this algorithm, each day is considered for buy and sell, and hence we're recursively computing the profit, comparing the stock prices of two days at a time, resulting in a time complexity of $O(n^2)$ for the complete day comparison. Further, for every buy scenario, we are considering all the stocks from 1 to m for buying, resulting in $O(m * n^2)$ complexity. Thus, the overall time complexity is $O(m * n^2)$.

Space Complexity: The algorithm is using a list to store the stock, buy day, and sell day indices. Since at maximum there would be k entries in this list which is equivalent to the number of transactions. Hence, the complete space complexity of the algorithm is O(k).

Algorithm 7 Alg7: Design a $O(m * 2^n)$ time brute force algorithm for solving Problem3

```
function BRUTEFORCE(stockIdx, buyIdx, dayIdx, canSell, txnList)

    maxProfit ← 0
    finalTxnList ← []

    if (dayIdx >= priceMatrix[0].length) then
        return {maxProfit, txnList}
    end if

    ▷ We already have a stock ◁
    if (canSell == true) then

        ▷ Holding the stock ◁
        {tmpMaxProfit, tmpTxnList} ← bruteForce(stockIdx, buyIdx, dayIdx + 1, canSell, txnList)
        currProfit ← tmpMaxProfit
        currTxnList ← tmpTxnList

        if (currProfit > maxProfit) then
            maxProfit ← currProfit
            finalTxnList ← currTxnList
        end if

        ▷ Selling the stock ◁
        diff ← priceMatrix[stockIdx][dayIdx] − priceMatrix[stockIdx][buyIdx]
        modifiedTxnList ← txnList
        modifiedTxnList.add([stockIdx, buyIdx, dayIdx])

        {tmpMaxProfit, tmpTxnList} ← bruteForce(null, null, dayIdx + cooldown +
        1, false, modifiedTxnList)

        currProfit ← tmpMaxProfit
        currTxnList ← tmpTxnList
        currProfit ← currProfit + diff

        if (currProfit > maxProfit) then
            maxProfit ← currProfit
            finalTxnList ← currTxnList
        end if

    else                                                        ▷ We don't have a stock
        ▷ Skip the day ◁
        {tmpMaxProfit, tmpTxnList} ← bruteForce(null, null, dayIdx + 1, false, txnList)
        currProfit ← tmpMaxProfit
        currTxnList ← tmpTxnList

        if (currProfit > maxProfit) then
            maxProfit ← currProfit
            finalTxnList ← currTxnList
        end if
```

```
        ▷ Buy stock of one of the companies ◁
        for i = 0 to priceMatrix.length do
            {tmpMaxProfit, tmpTxnList} ← bruteForce(i, dayIdx, dayIdx + 1, true, txnList)
            currProfit ← tmpMaxProfit
            currTxnList ← tmpTxnList

            if (currProfit > maxProfit) then
                maxProfit ← currProfit
                finalTxnList ← currTxnList
            end if

        end for

    end if

    return {maxProfit, finalTxnList}
end function
```

2.3.4 Dynamic Programming $O(m * n^2)$ - Algorithm

Recursive Formulation:
OPT(i) = (0 for OPT(0)
max{OPT(i -1), max{max{price[s][i] – price[s][j]} + OPT(j) }}

for all s and for all j in range [0, i-1] Otherwise

OPT(i) denotes the maximum possible profit obtained by selling on day i.

The recursive function has 3 cases:

Case 1: Profit = 0
We do not make any profit when day i = 0

Case 2: Profit = OPT(i -1)
OPT(i) does not select day i. So, there must be an optimal solution for days 1 to (i-1)

Case 3: Profit = max{max{price[s][i] – price[s][j]} + OPT(j) }
1. We first calculate the max{price[s][i] – price[s][j]} for all values of s
2. We select the best profit from OPT(j) to the maximize the profit calculated in step 1 for all j in range [0, i-1]

Figure 3: Recurrence Relation

Algorithm 8 Alg8: Design a $O(m * n^2)$ time dynamic programming algorithm for solving Problem3

```
function DPN2(priceMatrix, noOfDays, noOfStocks, cooldown)

    profit ← 1D array of size noOfDays
    stock ← 1D array of size noOfDays
    buy ← 1D array of size noOfDays

    profit[0] ← 0
    stock[0] ← 0
    buy[0] ← 0

    for dayloop = 1 to noOfDays do
        maxTotalCurrProfit ← 0
        buyIndex ← Integer.MINVALUE
        buyStockIndex ← 0
        for buyloop = 0 to dayloop do

            previousSellDay ← ((buyloop − cooldown − 1) < 0?0 : (buyloop − cooldown − 1))

            prevProfit ← profit[previousSellDay]
            maxcurrProfit ← 0
            stockIndex ← 0
            for stockloop = 0 to noOfStocks do

                currProfit ← priceMatrix[stockloop][dayloop] − priceMatrix[stockloop][buyloop]
                stockIndex ← (currProfit < maxcurrProfit?stockIndex : stockloop)
                maxcurrProfit ← max(currProfit, maxcurrProfit)
            end for

            profitHere ← maxcurrProfit + prevProfit
            if maxTotalCurrProfit < profitHere then
                buyIndex ← buyloop
                buyStockIndex ← stockIndex
            end if

            maxTotalCurrProfit ← max(maxTotalCurrProfit, profitHere)
            if profit[dayloop − 1] < maxTotalCurrProfit then
                stock[dayloop] ← buyStockIndex + 1
                buy[dayloop] ← buyIndex
            else
                stock[dayloop] ← stock[dayloop − 1]
                buy[dayloop] ← buy[dayloop − 1]
            end if

            profit[dayloop] ← max(profit[dayloop − 1], maxTotalCurrProfit)
        end for

    end for
```

```
    finalAns ← 2D list

    for dayloop = noOfDays − 1 to 1 do
        if (profit[dayloop] == profit[dayloop − 1]) then
            continue
        end if

        finalAns.add(stock[dayloop], buy[dayloop] + 1, dayloop + 1)
        dayloop ← buy[dayloop] − (cooldown + 1) + 1
    end for

    return finalAns
end function
```

2.3.5 Dynamic Programming $O(m * n^2)$ - Proof of Correctness

We would prove the correctness by induction –

- **Assertion** – At the end of nth (dayloop) iteration, the profit[dayloop] would hold the maximum profit.
- **Base Case** – When dayloop = 1, the profit[dayloop] would correctly return the maximum profit by buying at day0 and selling at day1 for the stock which returns the max profit.
- **Inductive Hypothesis** – Let's assume that algorithm gives maximum profit correctly for profit[dayloop – 1] for the (dayloop – 1)th iteration, we would try to prove that it provides the correct output for profit[dayloop] for the dayloopth iteration.
- **Inductive Step** –
  • We calculate the maximum profit that can be made on the last day (dayloop = n) for all the previous buy days and stocks. • We perform summation of previous profit value at profit[buyloop -cooldown -1] and the max profit that can be made at dayloop max(priceMatrix[stockloop][dayloop] - priceMatrix[stockloop][buyloop]) where buyloop is (0 to dayloop -1) and stockloop is (0 to total number of stocks). We store this maximum profit for the current iteration of dayloop (last day) in maxTotalCurrProfit.
  • We calculate profit[dayloop] = max(profit[dayloop - 1], maxTotalCurrProfit) which takes the maximum profit between the profit calculated in (dayloop – 1)th iteration and the dayloopth iteration.
  • Thus, our assertion holds true that the maximum profit is returned.

2.3.6 Dynamic Programming $O(m * n^2)$ - Time and Space Complexity Analysis

Time Complexity: There are three loops, the outermost loop iterating on the days, ranging from 1 to n-1 and having a complexity O(n). The next loop is a loop on the buy days going from 0 to the day in the first loop having a worst-case complexity of O(n). The last loop is iterating on the number of stocks from 1 to m having a complexity O(m). Hence, the overall time complexity of the three nested loops is $O(m * n^2)$

Space Complexity: The algorithm is using three 1D arrays to store the stock, buy day, and sell day indices, each of size (n). Hence, the complete space complexity of the algorithm is O(n)

2.3.7 Dynamic Programming $O(m * n)$ - Algorithm

Recursive Formulation:
OPT(i) = (0 for OPT(0)
max{OPT(i -1), max{ price[s][i], max{OPT(j) – price[s][j] } } }

for all s and for all j in range [0, i-1] Otherwise

OPT(i) denotes the maximum possible profit obtained by selling on day i

The recursive function has 3 cases:

Case 1: Profit = 0
We do not make any profit when day i = 0

Case 2: Profit = OPT(i -1)
OPT(i) does not select day i. So, there must be an optimal solution for days 1 to (i-1)

Case 3: Profit = max{ price[s][i], max{OPT(j) – price[s][j] } }

1. Take the price[s][i] for all values of s
2. We are recursively executing OPT(j) for all j in range [0, i-1]. We take the maximum of {OPT(j) – price[s][j]}
3. Taking the maximum value returned from step 2, we would add the price from step 1 to it and check for all stocks to get the maximum profit

Figure 4: Recurrence Relation

2.3.8 Dynamic Programming $O(m * n)$ - Proof of Correctness

We would prove the correctness by induction –

- **Assertion** – At the end of nth (dayloop) iteration, the $profit[dayloop]$ would hold the maximum profit.
- **Base Case** – When dayloop = 1, the profit[dayloop] would correctly return the maximum profit by buying at day0 and selling at day1 for the stock which returns the max profit.
- **Inductive Hypothesis** – Let's assume that algorithm gives maximum profit correctly for $profit[dayloop – 1]$ for the (dayloop – 1)th iteration, we would try to prove that it provides the correct output for $profit[dayloop]$ for the dayloop-th iteration.
- **Inductive Step** –
  We have already calculated the maximum previous difference until (dayloop - 2) where previous difference is defined as $max(profit[j - 1 - cooldown - 1] – priceMatrix[j])$ where j is (0 to dayloop -2).
  • We calculate the previous difference for (dayloop -1) and store it in the variable $yesterdaysDiffwithProfit$.
  We calculate $prevDiffWithProfit[stockloop] = max(prevDiffWithProfit[stockloop], yesterdaysDiffwithProfit)$ which stores the maximum previous difference until (dayloop - 1) and stockloop is (0 to total number of stocks).
  • We calculate the maximum profit that we get by comparing the $profit[dayloop – 1]$ with the $profit[dayloop][stockloop]$ where stockloop is (0 to total number of stocks). We store the maximum returned in $profit[dayloop]$.
  • Thus, the profit for $profit[dayloop]$ takes the maximum profit between the profit calculated in (dayloop – 1)th iteration and the dayloop-th iteration.
  • Thus, our assertion holds true that the maximum profit is returned.

Algorithm 9 Alg9: Design a O(m * n) time dynamic programming algorithm for solving Problem3

```
function DPN(priceMatrix, noOfDays, noOfStocks, cooldown)

    profit ← 1D array of size noOfDays
    stock ← 1D array of size noOfDays
    buy ← 1D array of size noOfDays
    prevDiffWithProfit ← 1D array of size noOfStocks
    buyIndex ← 1D array of size noOfStocks

    profit[0] ← 0
    stock[0] ← 0
    buy[0] ← 0

    for stockloop = 0 to noOfStocks do
        prevDiffWithProfit[stockloop] ← Integer.MINVALUE
        buyIndex[stockloop] ← 0
    end for

    for dayloop = 1 to noOfDays do

        if (dayloop − 1 − cooldown − 1) < 0 then
            previousDay ← 0
        else
            previousDay ← (dayloop − 1 − cooldown − 1)
        end if

        yesterdaysProfit ← profit[previousDay]

        for stockloop = 0 to noOfStocks do
            yesterdaysDiffwithProfit ← yesterdaysProfit − priceMatrix[stockloop][dayloop − 1]

            if (prevDiffWithProfit[stockloop] <= yesterdaysDiffwithProfit) then
                buyIndex[stockloop] ← dayloop − 1
            end if

            prevDiffWithProfit[stockloop] ← max(prevDiffWithProfit[stockloop], yesterdaysDiffwithProfit)

            tempMax ← max(profit[dayloop − 1], priceMatrix[stockloop][dayloop] +
            prevDiffWithProfit[stockloop])

            if (tempMax > profit[dayloop]) then
                stock[dayloop] ← stockloop + 1
                buy[dayloop] ← buyIndex[stockloop]
            end if

            profit[dayloop] ← max(tempMax, profit[dayloop])
        end for

    end for
```

$finalAns \leftarrow$ 2D list of integers

for $dayloop = k\ to\ noOfDays - 1$ **do**
  if $(profit[dayloop] == profit[dayloop - 1])$ **then**
    continue
  end if

  $finalAns.add(stock[dayloop], buy[dayloop + 1], dayloop + 1)$
  $dayloop \leftarrow buy[dayloop] - (cooldown + 1) + 1$
end for

return $finalAns$
end function

2.3.9 Dynamic Programming $O(m * n)$ - Time and Space Complexity Analysis

Time Complexity: There are two loops, the outermost loop iterating on the days, ranging from 1 to n-1 and having a complexity O(n). The second loop is iterating on the number of stocks from 1 to m having a complexity O(m). Hence, the overall time complexity of the nested loops is $O(m * n)$

Space Complexity: The algorithm is using five 1D arrays to store the stock, buy day, sell day indices, profit, and previous difference with profit, each of size (n). Hence, the complete space complexity of the algorithm is O(n)

3 Experimental Comparative Study

3.1 Problem1

From plots 1 and 2, we observe that the brute force approach is the slowest and the time taken by it on larger input sizes increases extremely quickly. Greedy approach seems to always be the fastest. On inputs of larger sizes, greedy approach outdoes all the other approaches considerably. DP with memoization is almost two times slower than bottom-up DP. Both the DP approaches give a stack overflow error on large inputs.

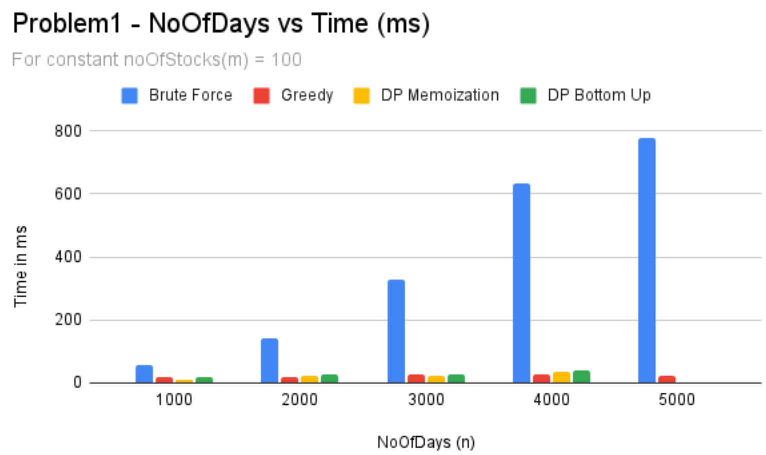

Figure 5: Plot1 - Comparison of Task1, Task2, Task3A, Task3B with variable n and fixed m

Problem1 - Plot1 Table					
m	n	Brute Force	Greedy	DP Memoization	DP Bottom Up
100	1000	55	20	9	19
100	2000	143	20	24	28
100	3000	327	29	25	28
100	4000	632	29	34	41
100	5000	777	25	Stack Overflow	Stack Overflow

Figure 6: Table for Plot1

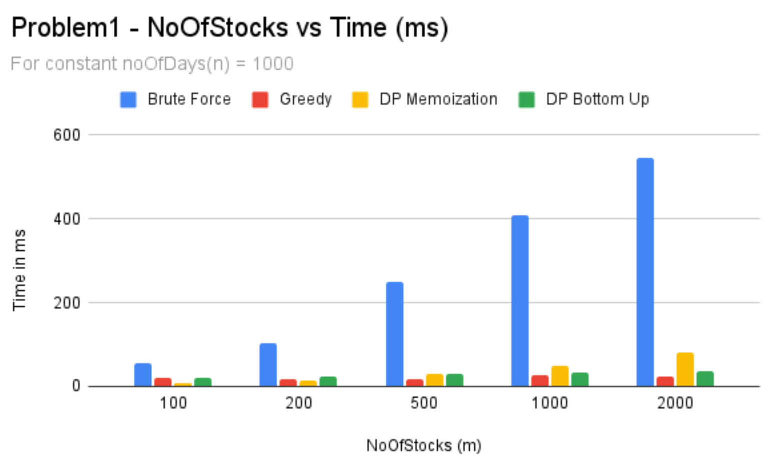

Figure 7: Plot2 - Comparison of Task1, Task2, Task3A, Task3B with variable m and fixed n

Problem1 - Plot2 Table					
m	**n**	**Brute Force**	**Greedy**	**DP Memoization**	**DP Bottom Up**
100	1000	55	20	9	19
200	1000	104	17	15	23
500	1000	251	18	31	29
1000	1000	408	26	50	34
2000	1000	545	24	81	38

Figure 8: Table for Plot2

3.2 Problem2

From plots 3,4,5 and the Brute Force plots, we observe that the bottom-up dynamic programming approach is the fastest even when we vary the inputs for the number of stocks, number of transactions, and number of days, while keeping the other inputs fixed. The second fastest is the DP memoization approach, whose values as seen in the graph are the next closest to the DP bottom-up approach. Complexity and timewise, the iterative dynamic programming approach having $O(m * n^2 * k)$ takes a long time to compute the best buy and sell indices and maximum profit. Brute force takes the longest time to return the maximum profit, and it had to be tested against data that is relatively smaller as compared to the n, m, and k values for testing the other programming paradigm approaches.

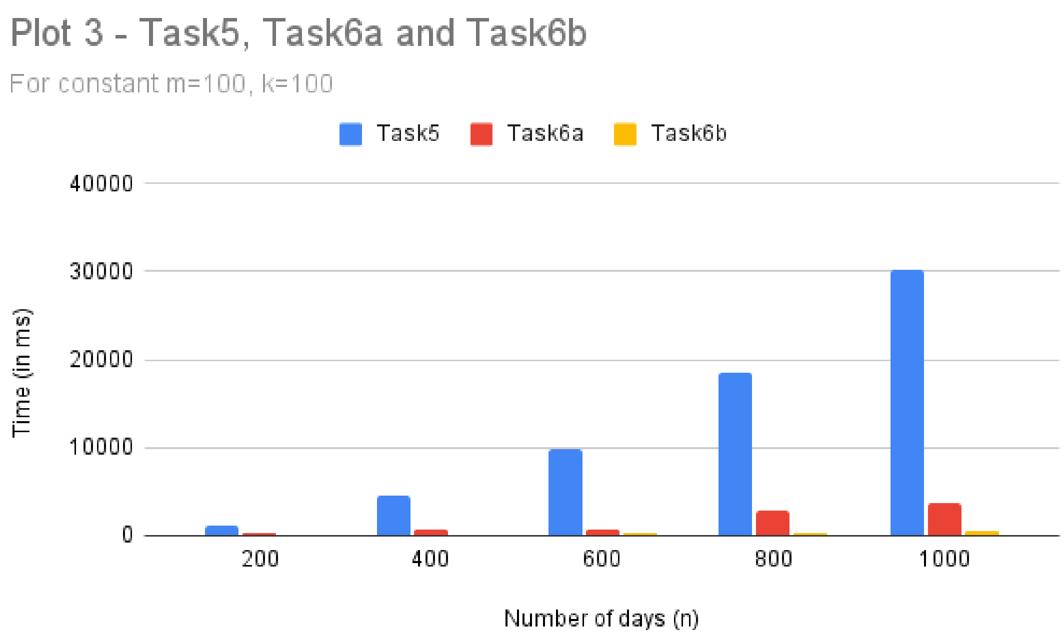

Figure 9: Plot3

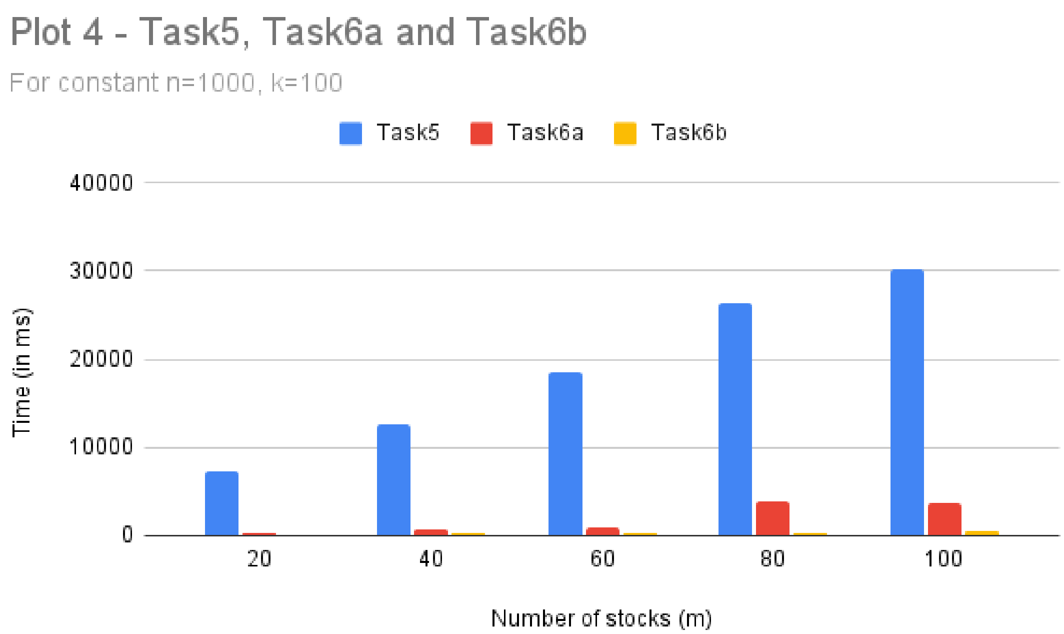

Figure 10: Plot4

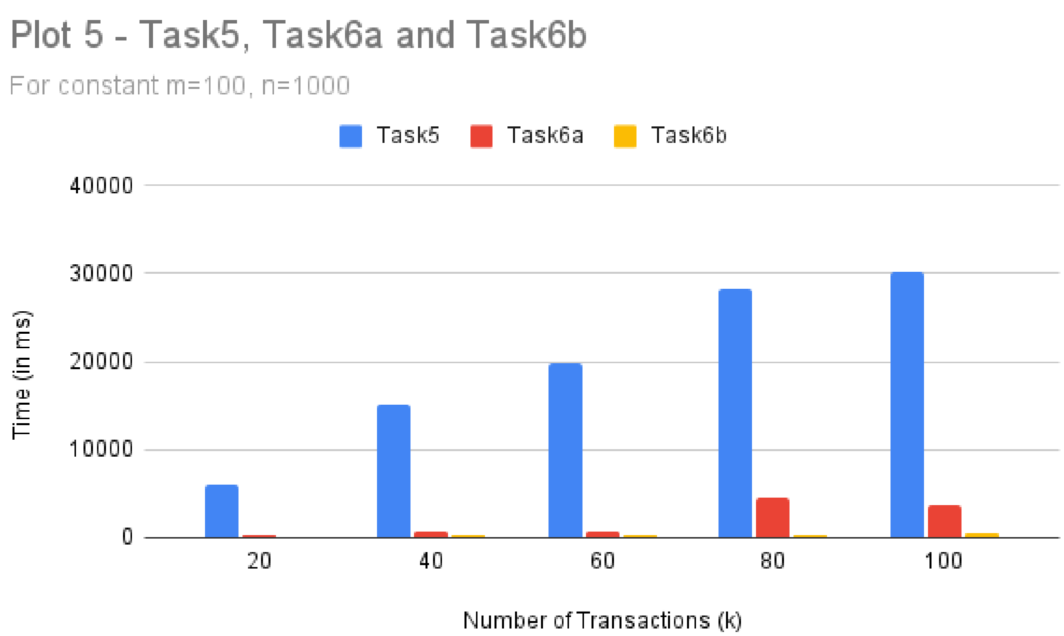

Figure 11: Plot5

Problem2 - Plot3 Table (Variable n)					
m	n	k	Task5	Task6a	Task6b
100	200	100	1184	240	113
100	400	100	4556	729	192
100	600	100	9915	826	228
100	800	100	18586	2820	245
100	1000	100	30100	3803	442
Problem2 - Plot4 Table (Variable m)					
m	n	k	Task5	Task6a	Task6b
20	1000	100	7249	259	84
40	1000	100	12603	736	223
60	1000	100	18478	935	293
80	1000	100	26328	3982	336
100	1000	100	30100	3803	442
Problem2 - Plot5 Table (Variable k)					
m	n	k	Task5	Task6a	Task6b
100	1000	20	5933	307	159
100	1000	40	15210	658	225
100	1000	60	19879	814	228
100	1000	80	28317	4452	361
100	1000	100	30100	3803	442

Figure 12: Table for Plots 3, 4, 5

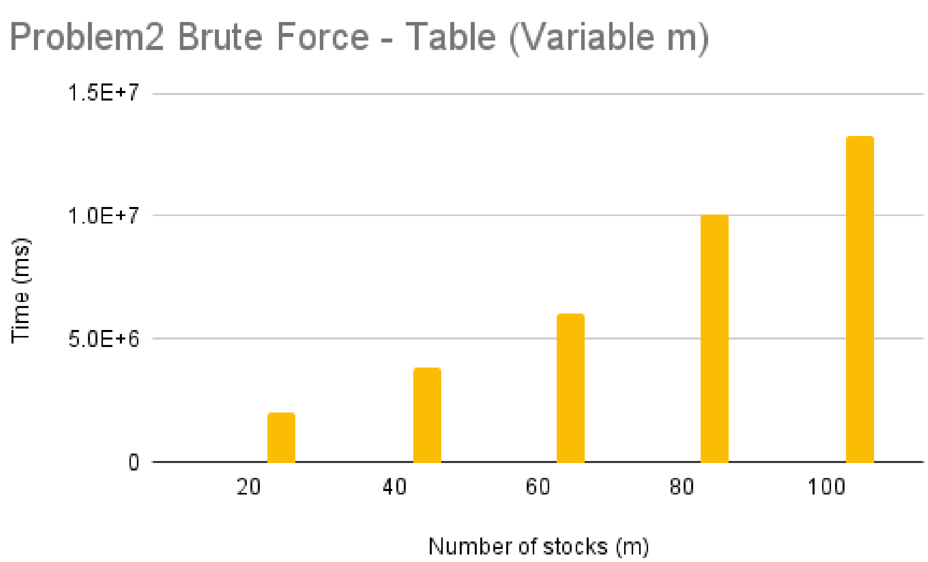

Figure 13: Brute Force

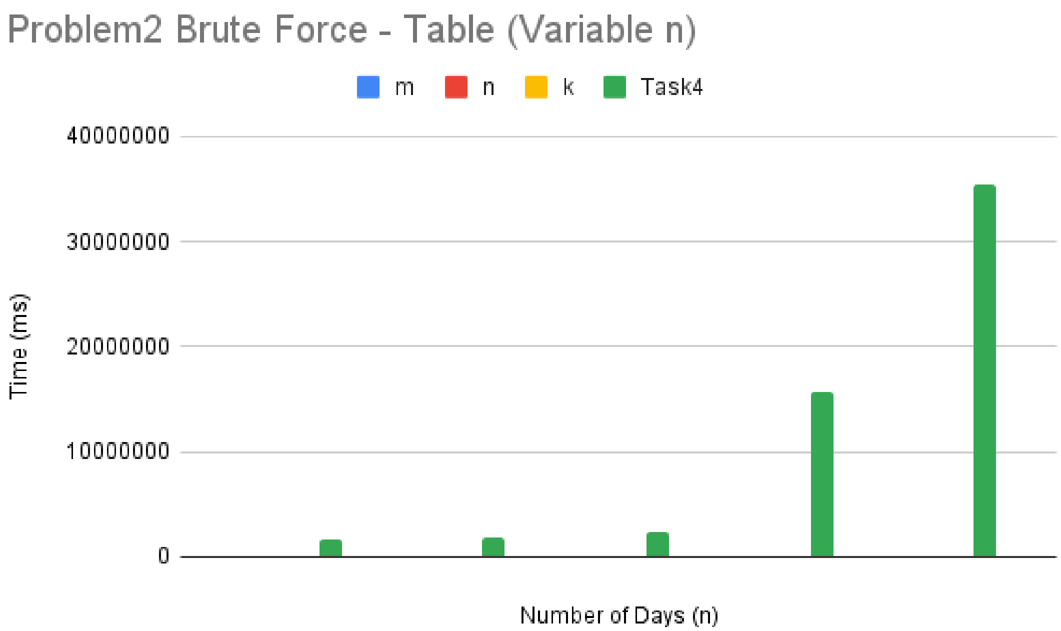

Figure 14: Brute Force

Problem2 - Table (Variable m)			
m	n	k	Task4
20	50	5	2018946
40	50	5	3825643
60	50	5	6032452
80	50	5	10037837
100	50	5	13256482
Problem2 - Table (Variable n)			
m	n	k	Task4
50	20	5	1723145
50	40	5	1943267
50	60	5	2467458
50	80	5	15735463
50	100	5	35346789

Figure 15: Table for Problem2 Brute Force

3.3 Problem3

From plots 8,9 and the Brute Force plots, we can infer that the brute force algorithm is taking the longest time to run, followed by Task 8 in Problem 3, where the complexity is $O(m * n^2)$. The dynamic programming memoization and bottom-up approaches are running relatively faster than the other algorithms. The bottom-up approach runs the fastest for different variable m and fixed n values and variable n and fixed m values. The memorization algorithm with a complexity of O(m*n) can be observed to be the second fastest with variable m and fixed n values and vice versa as well.

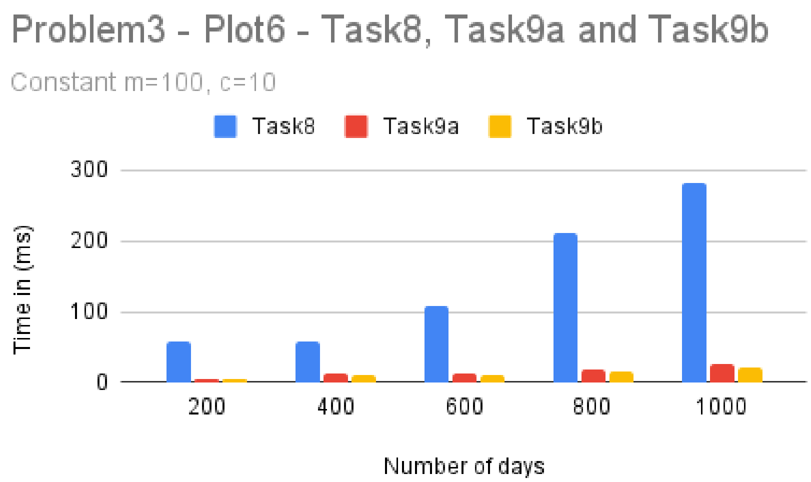

Figure 16: Plot6

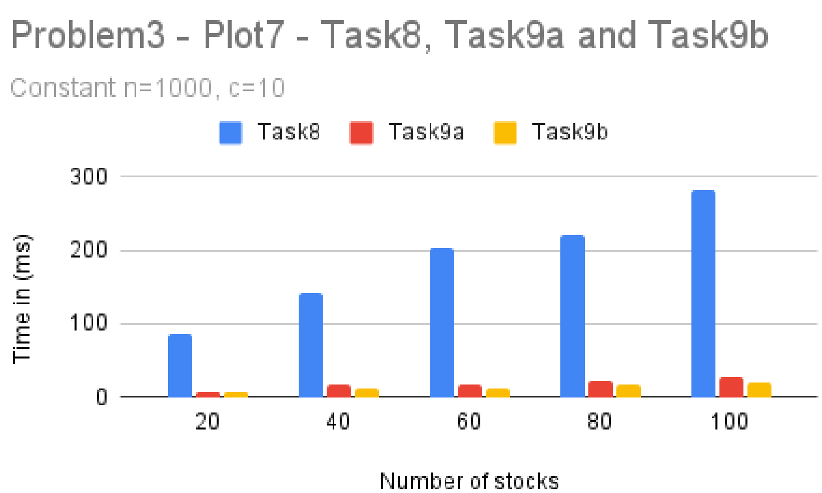

Figure 17: Plot7

Problem3 - Table (Variable n)					
m	n	c	Task8	Task9a	Task9b
100	200	10	58	5	5
100	400	10	59	13	10
100	600	10	107	12	11
100	800	10	210	18	15
100	1000	10	281	27	20

Problem3 - Table (Variable m)					
m	n	c	Task8	Task9a	Task9b
20	1000	10	86	7	6
40	1000	10	142	16	11
60	1000	10	203	18	13
80	1000	10	222	23	18
100	1000	10	281	27	20

Figure 18: Table for Plots 6 and 7

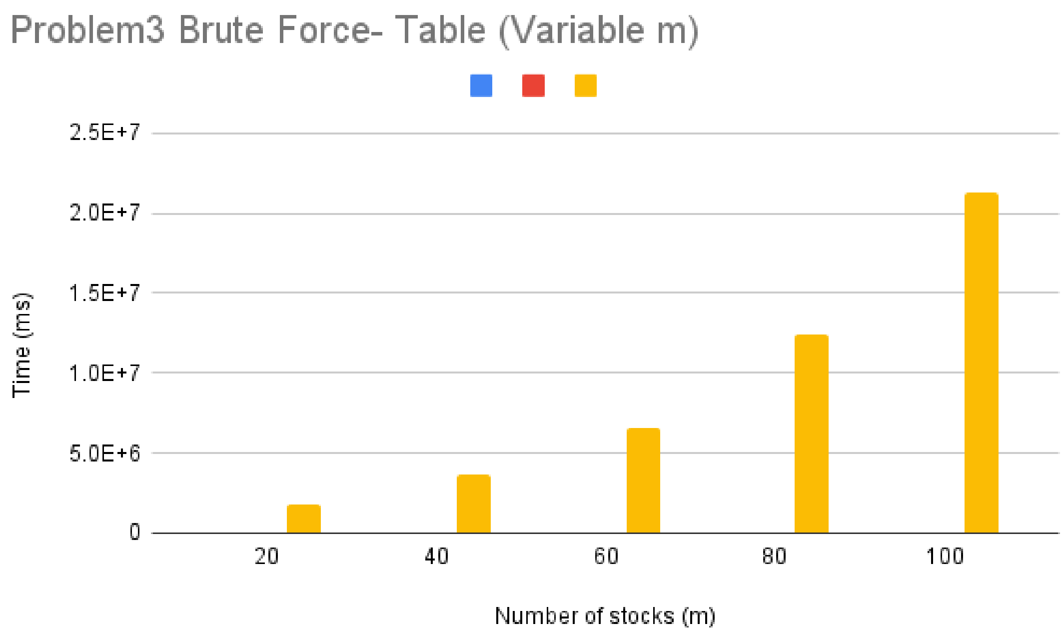

Figure 19: Brute Force

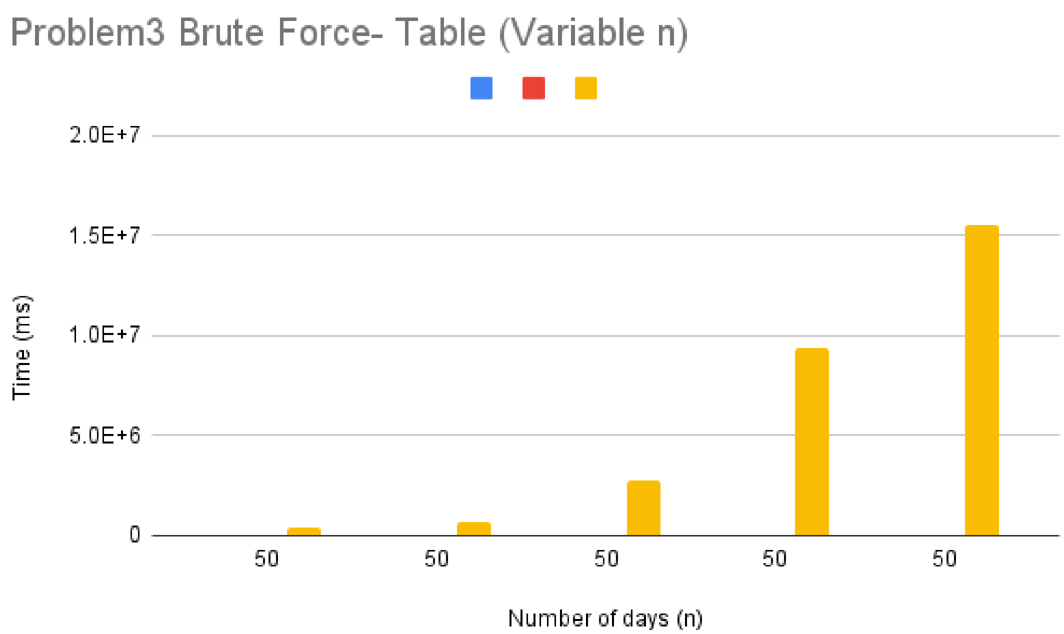

Figure 20: Brute Force

Problem3 - Brute Force Table (Variable m)			
m	n	c	Task7
20	50	10	1847856
40	50	10	3688234
60	50	10	6573122
80	50	10	12446232
100	50	10	21346294
Problem3 - Brute Force Table (Variable n)			
m	n	c	Task7
50	20	10	456900
50	40	10	667890
50	60	10	2756746
50	80	10	9343654
50	100	10	15562366

Figure 21: Table for Problem3 Brute Force

4 Conclusion

Task1: This was the easiest among all tasks. It was not very difficult to come up with a brute force solution for this problem. Coding it was also very easy. But it had the worst time complexity of all approaches since it was blindly trying all possible solutions.

Task2: For problem1, coming up with a greedy strategy was tricky. The greedy algorithm is the best in terms of time and space complexity. It doesn't use any extra space except for a few integer variables and doesn't use recursion so there is no chance of a stack overflow error. The greedy algorithm could handle inputs of large sizes and was always the quickest. This shows that greedy is always a better choice over brute force or dynamic programming in terms of performance if a correct greedy algorithm exists for the problem.

Task3a: This was the DP with memoization task. We observed that it was up to x2 slower than bottom-up DP. This was because of all the recursive calls it was making. For larger input sizes, this approach started giving a stack overflow error because of all the space being used up in recursive calls and creation of memoization table.

Task3b: It was easy to come up with this recurrence as we have only 1 transaction here. The implementation was easy with 2 nested loops and debugging was also simpler to achieve.

Task 4: This seems easier at first glance but implementation was a little tricky as we were comparing the maximum profit for all combinations of m, n and k. Debugging the code seemed to be challenging due to high number of comparisons getting made.

Task 5: The recurrence relation was simpler to write as we are iterating over N twice. The implementation was relatively complex as we had to deal with 4 nested loops.

Task 6a: It was challenging to come up with the recurrence relation while traversing through N only once. The memoized implementation was easy to implement as it used recursion instead of loops. Debugging was challenging as it was difficult to pin point which iteration of the recursion was yielding the result.

Task 6b: It was challenging to come up with the recurrence relation while traversing through N only once. Once the recurrence relation was finalized, it was easier to implement than Task 5, as we have smaller number of iterations here.

Task 7: This was relatively easy to implement as this is an extended version of Task 4 with infinite transactions and an additional cooldown period to take care of. Debugging was still a challenge here, due to the high number of comparisons getting made.

Task 8: This was relatively easy to implement as this is an extended version of Task 5 with infinite transactions and an additional cooldown period to take care of. This implementation had 3 nested loops which made it slightly less complex in terms of coding.

Task 9a: This was relatively easy to implement as this is an extended version of Task 6 with infinite transactions and an additional cooldown period to take care of. This implementation was easier with decreased number of recursions. Debugging was still posed a challenge.

Task 9b: This was relatively easy to implement as this is an extended version of Task 6 with infinite transactions and an additional cooldown period to take care of. This implementation was the easiest, with decreased number of loops/recursions and debugging was also easy to handle.