COT5405 ANALYSIS OF ALGORITHMS

Programming Project 2022

Team Members And Contribution -

1.Ekta Bhaskar - Devised algorithm for problem 2 & Bonus, implemented (UFID:71592221) code for problem 2.

2.Piyush Singh - Devised algorithm for problem 1& Bonus, implemented (UFID:50927342) code for problem 1.

3.Ayush Shrivastava - Devised algorithm for problem 1 & 2, implemented (UFID:41218133) code for Bonus.

Combined Contribution: Comparative analysis, Generating test cases, Proof of correctness, time and space analysis.

Design and Analysis of Algorithms -

Task 1

Algorithm

The algorithm used for task1 is the Bruteforce algorithm. We are using 3 nested loops to iterate through the 2D matrix of m stocks for n days while storing and updating the pairs of stocks to buy and sell. We attained this through one transaction by buying the stock each day and selling it for the current max profit and then comparing it with each stock and returning the max_profit.

PseudoCode

```
Task 1(StocksPriceMatrix [m][n])
        Max_profit  ← 0
        Fin_stock   ← -1
        Fin_buy     ← -1
        Fin_sell    ← -1

        For i = 0 → m.lenght()
```

```
For j = 0 → n.lenght()
    For k = j+1 → n
        IF StocksPriceMatrix[i][k] > StocksPriceMatrix[i][j]
          THEN curr_diff = StocksPriceMatrix[i][k] - StocksPriceMatrix[i][j]
          IF curr_diff > Max_profit
            THEN  Max_profit  ← 0
                  Fin_stock    ← i
                  Fin_buy      ← j
                  Fin_sell     ← k
RETURN (Max_profit , Fin_stock, Fin_buy,  Fin_Sell)
```

Time Complexity

As we are looping through 3 nested loops ranging from m, n, and n respectively, the time complexity for this task is O(m*n*n) = O(m*n^2).

Space Complexity

As no additional Space is required. Auxiliary space complexity is O(1)

Correctness

Initialization :

- The answer will hold the max profit and its initialized as 0.

Maintenance:

- In each nested iteration, we iterate through the different stock prices each day then iterate all the stocks similarly while storing the Current_max by buying and selling stocks for maximum profit.
- If a difference is larger than Current_max we keep on updating it till termination.
- This is Brute force algo so every buy-sell combination is exhausted to attain max_profit.

Termination:

- The nested loops terminate once we have exhausted all the profit combinations of the 2D stocks array and the current_max is the largest profit to return as the answer.

Task 2

Algorithm

The algorithm used for task 2 is the Greedy Algorithm.
For this, we are using 2 nested loops to iterate through the stock prices of different companies. Along the process, we are storing the lowest price we have seen so far and also the biggest profit we can achieve respective to each iteration previously done. Then we store the profit in current_profit and keep on referencing it till a bigger profit is achieved, then we update it. This will achieve us the max_profit and the best answer so far. We are using the variable mini to save the minimum stock price so far.

PseudoCode

```
Task 2(StocksPriceAtDay [m][n])
        Max_profit  ← Integer.MIN_VALUE
        Mini         ← Integer.MAX_VALUE
        Fin_stock    ← -1
        Fin_buy []     ← Array of size(m)          #To save the min price found of stock i
        Fin_sell  []   ← Array of size(m)          #To save the max price found of stock i

        For i = 0 → m.lenght()
            Curr_profit  ← Interger.MIN_VALUE

            For j = 0 → n.lenght()
                    IF StocksPriceAtDay[i][j] < mini
                       THEN mini = StocksPriceAtDay[i][j]
                               Fin_buy[i] = j
                    ElseIF StocksPriceAtDay[i][j] - mini > Curr_profit
                          THEN Curr_profit = StocksPriceAtDay[i][j] - mini
                          IF Curr_profit > max_profit
                             THEN Fin_stock ←  i
                                     Max_profit ← curr_profit
                          Fin_sell[i] ← j


        RETURN (Max_profit , Fin_stock + 1 , Fin_buy [Fin_stock], Fin_Sell [Fin_stock])
```

Time Complexity

As we are looping through 2 nested loops ranging from m and n respectively, the time complexity for this task is O(m*n) .

Space Complexity

Since two auxiliary of size m were used to store the min and max stock values while iterating. Hence space complexity is O(2*m).

Correctness

Initialization :

- The answer will hold the max_profit and its initialized as Integer.MIN_VALUE.
- Variable mini is also initialized as Integer.MAX_VALUE.It is being used to retrieve the lowest value seen so far.

Maintenance:

- In each nested iteration, we are looping through m stocks whose prices are changing every nth day. Meanwhile, compare the ith stock price with mini and update mini if found a smaller value and store the jth day to buy that stock in fin_buy.
- We are also maintaining the largest profit yet in curr_profit = stockprice[i][j] - mini.
- Furthermore updating curr_profit with max_profit(ans).
- This is a Greedy algorithm so while maintaining the best answer so far just return the answer in the end.

Termination:

- The nested loops terminate once we have iterated through all the stocks while storing and updating the max_profit and mini variables.

Task 3a

Algorithm

Here we are using the same basic principle of finding maximum profit transactions and comparing values of all stocks to get max_profit using 2 helper_functions as discussed further. Helper_function1 is used to find the maximum profit transaction for each stock which is done recursively and then helper_function2 recursively saves it in the stock_wise_transaction[stock][day] matrix for storing the current max profit by storing buy and sell date for every stock. In helper_function2 we are filling the memo table for each stock and the memo contains cur_day_transaction and last_day_transaction. The memoization using stock_wise_transaction increasing time-complexity efficiency.

Recurrence Relation

$OPT\{priceAtDay,\ DP,\ stock,\ day\} \longrightarrow$

IF (day == 0) $OPT\{priceAtDay,\ DP,\ stock,\ day\}$
Else $OPT\{priceAtDay,\ DP,\ stock,\ day-1\}$

The recurrence relation in helper function is used to determine the selling day of stock to maximise the profit for each stock and it is being memoized in DP table.

PseudoCode

Transaction* (Class for object oriented design)

```
Helper_function1 (price_at_day[][] , DP[][] , stock)
                helper_function2(price_at_day[][] , DP[][], stock, price_at_day[0].len - 1 )
                Max_for_stock ← DP[stock][0]
                FOR day = 0 → n.length()
                     Max_for_stock ← max(Max_for_stock , DP[stock][day])
                Return max_for_stock

Helper_function2 (price_at_day[][] , DP[][] , stock, day)
          IF day = 0
            DP[stock][day] = new Transaction(stock, day, day, 0(profit:))
            Return DP[stock][day]
          IF DP ! null
            Return DP[stock][day]
          Transaction cur_day_T ← new Transaction(stock, day, day, 0(profit:))
          Transaction last_day_T ← Helper_function2(price_at_day, DP, stock, day-1)
          Cur_day_T.profit ← last_day_T.profit  - (price_at_day[stock][day]
→                                                    -price_at_day[stock][day-1])
          Return DP[stock][day]
```

Task 3a

(Price_At_Day [m][n])

```
        Transaction max_transaction = new Transaction(
                                                        Max_profit  ← 0
                                                        Fin_stock    ← -1
                                                        Fin_buy      ← -1
                                                        Fin_sell      ← -1 )

        Transaction DP[][] ← new Transaction[m][n]

            For stock = 0 → m.length()
                Max_transaction = max(max_transaction , helper_func1(price_at_day, DP,
→                                                                          stock))
            Return(max_transaction.getStockNumber() + 1, max_transaction.getBuyDay
→                       + 1, max_transaction.getSellDay() + 1, max_transaction.getProfit() )
```

Time Complexity

The time complexity for the DP memoization based solution in O(m*n).

Space Complexity

Since two auxiliary of size m*n matrix were used to store the largest profit. Hence space complexity is O(m*n).

Correctness

Initialization :

- The answer will hold the max profit and its initialized as 0.

Maintenance:

- In the iteration stock[0,...m] we are comparing max_transaction with the helper_function that is recursively comparing the same fo each stock.
- Helper_function1 is calling to helper_function2 to fill up the DP[][] 2D array to remove the redundant work that is already performed and then comparing the max_for_stock with DP[stock][day] to return the max_for_stock.
- Helper_Function2 is calculating the max transaction and filling the memo table and if the value is non null return it directly.

Termination:

- The helper_functions terminate whenever the 2D array is recursively iterated and the max_transaction is returned.
- In helper_function all the edge cases are implemented to increase the time efficiency of algorithm.

Task 3b

Algorithm

Here we are using bottom-up DP algorithm.The max_profit is attained by calculating the max consecutively increasing difference.The maximum consecutively increasing difference can end at any element in nums.The basic logic involves the maximum consecutively increasing difference will end at (i+1)th position which means it either includes MCI at position i or it doesn't. At the end max_profit is returned and the progress is being tabulated using DP array maxIncDiff[][].

Recurrence Relation

i=0 ->m
j=0 ->n

IF $maxIncDiff[i][j] \ + \ \{priceAtDay[i][j] - ... [i][j-1]\} >= 0 \longrightarrow$

$$maxIncDiff[i][j] = maxIncDiff[i][j] \ + \ \{priceAtDay[i][j] - ... [i][j-1]\}$$

Else $maxIncDiff[i][j] = 0$

The recurrence relation in DP function is used to determine the maximum consecutively increasing difference to maximise the profit for each stock and it is being stored in DP table.

PseudoCode

Transaction* (Class for object oriented design)

Task 3b

```
(Price_at_day [m][n])
        Max_profit        ← Integer.MIN_VALUE
        MaxIncDiff [][]   ← [m][n]
        Fin_buy[]          ← m.length
        Fin_sell           ←  m.length
        Fin_stock          ← -1

        For i = 0 → m.lenght()
            For j = 0 → n.lenght()
                IF maxIncDiff [i][j-1] + price_at_day [i][j] - price_at_day [i][j-1] >= 0
                  THEN maxIncDiff [i][j] = maxIncDiff [i][j-1] + price_at_day [i][j] -
→                                                              price_at_day [i][j-1]
                         Fin_sell ← j
                ELSE maxIncDiff [i][j] ← 0
                       Fin_buy ← j
                IF maxIncDiff [i][j] > max_profit
                  THEN max_profit ← maxIncDiff [i][j]
                         Fin_stock ← i

        Return new Transaction(fin_stock + 1, fin_buy[fin_stock] + 1, fin_sell[fin_stock] +
→                                    1, max_profit)
```

Time Complexity

The time complexity for the DP tabulation based solution in O(m*n).

Space Complexity

Since two auxiliary of size m*n matrix were used to store the largest profit. Hence space complexity is O(m*n).

Correctness

Initialization :

- The answer will hold the max profit and its initialized as 0.

Maintenance:

- The maxIncDiff[i][j] stores the max sum of increasing difference for stock i on day j.
- The fin_buy and fin_sell are stored to return the resulting max_profit.
- The DP 2D array is being used to max_sum and if it less than zero

Termination:

- The function terminates when all the edge cases when the whole DP array maxIncDiff[][] has been iterated.

Task 4

Algorithm

The concept used for question 2 is based on a Brute force recursive-based algorithm. In this we are using a helper_function for recursion which will return a tuple of maximum profit and sequence of.We are comparing the sell_state and the buy_state and both of which consists of 2 cases.

Sell state cases:

1. not sell on current day that means we still hold a stock, so the sell_state for next day will be true. And since we're not doing anything on current day, number of transactions will remain same.
2. by selling on current day, we're finishing 1 transaction, hence k = k-1. And then sell_state for next day will be false as we have to buy before selling again.

Buy state cases:

1.Don't buy on current day, so buy_state for next day will be true and no transactions are made, hence k remains unchanged.

2.Buy on cur day, then sell_state of next day will be true. A transaction is complete when a stock has been sold. Hence k remains unchanged.

Recurrence Relation

i=0 ->m
k=0 ->n

OPT(i,k) = $taskHelperDP(i\,, Bool\,,\ k\,,\ m)$!=0 {
$taskHelperDP(i\ +\ 1\,, Bool\,,\ k\,,\ m)$
$taskHelperDP(i\,, Bool\,,\ k\ -\ 1\,,\ m)$ }

The recurrence relation in helper function is used to determine the selling day of stock to maximise the profit for each stock and it is being in DP table.

PseudoCode

Transaction* (Class for object oriented design)

Task 4

(Price_At_Day[m][n] , k)

```
        priceAtDay4 ←  priceAtDay
        Res_tuple = task4_helper(i:0, sell: false, k, m:0 )
        printLn "Max profit is: " + res_tuple.getProfit()
        IF res_tuple.getTransactionList() == null    OR
→           res_tuple.getTransactionList().getTransactionList_L().isEmpty()
          THEN Println → "No Transactions are found to print, profit must be 0 and
                          hence no transaction list is empty"
        RETURN res_tuple.getTransactionList().getTransactionList_L()
```

Task4_Helper

```
  (int i, boolean sell, int k, int m)

  Len_stocks ← priceAtDay4.length()
  N ← priceAtDay4[0].length()
  IF k == 0
```

```
  THEN Return (0, newTransactionL())
IF i == n
  THEN Return (0, newTransactionL())
IF (sell)
   THEN
     Tuple1 = Task4_helper(i+1, true, k, m )
     Tuple2 = Task4_helper(i, true, k-1, m )
     Val1 ← Tuple1.getProfit()
     Val2 ← Tuple2.getProfit()
     X1 ← tuple1.getTransactionList()
     X2 ← tuple2.getTransactionList()
     IF val1 > val2 + priceAtDay4[m][i]
        THEN return (val1, x1.add(new Transaction(-m, -1, i, priceAtDay4[m][i]))
     RETURN (val2 + priceAtDay4[m][i] , x1.add(new Transaction(-m, -1, i,
                    priceAtDay4[m][i]))
ELSE
        Total-max ← 0
        List x3 ← new Arraylist()
        FOR m1 = 0 ← len_stocks
              Tuple1 = Task4_helper(i+1, false, k, m1 )
              Tuple2 = Task4_helper(i+1, true, k, m1 )
              Val1 ← Tuple1.getProfit()
              Val2 ← Tuple2.getProfit()
              X1 ← tuple1.getTransactionList()
              X2 ← tuple2.getTransactionList()
              Val2 ← val2 - priceAtDay4a[mi][i]
              List xf ← new Arraylist()
              List xf2 ← new Arraylist()
              FOR t: x2.getTransactionList_L()
                   IF t.getBuyDay == -1
                     THEN xf2.add(new Transaction(t.getStockNumber, i, t.getSellDay,
                                                    t.getProfit - priceAtDay4[m][i]))
                   ELSE xf.add(t)
              Temp ← Math.max(val1, val2)
              IF total_max <  temp
                THEN IF val1 == temp
                         THEN x3 = x1.getTransactionList_L()
                       ELSE xf.addAll(xf2.sublist(0, 1))
                              X3← Arraylist(xf)
                       Total_max = temp
```

Return tuple(total_max, TransactionL(x3))

Time Complexity

The time complexity for the Brute force recursive based solution in O($m * n^{2k}$).

Space Complexity

Since two auxiliary of size m*n matrix were used to store the largest profit. Hence space complexity is O(m*n).

Correctness

Initialization :

- The answer will return a res_tuple that is initialized as 0 which will return the set of transactions to calculate max_profit.
- A helper_funtion is being used to fill the res_tuple with buy and sell transactions.

Maintenance:

- Once we have calculated the buy and sell day state and stored it in different tuples.
- We calculate the profit for both the cases and store it in val1 and va2. And then storing the stocks in xf and xf2 which have a buy and and doesn't respectively.

Termination:

- The helper_functions terminate whenever the 2D array is recursively iterated and the max_transaction is returned.
- In helper_function all the edge cases are implemented to increase the time efficiency of algorithm.

Task 5

Algorithm

The DP algorithm used in problem 2 is tabulation and we are using the profitDP[][] to keep track of the k transactions in n days. We calculate the profit for buy on day x and selling it on jth day buy nested looping through the stocks and days and then checking in (j-1)th achieves the most profit till jth day.Returning the max profit by removing the overlaps from the transaction list.

Recurrence Relation

$OPT(i, j) \;=\; task5(j >= 1)$

DP_profit [i][j] **=** Max{ max1 , DP_profit [i][j-1]}

The recurrence relation for this algorithm is calculated because at each point maximum proft so far will be maximum of profit till (j-1)th day and maximum profit by selling on jth day for j >=0.

PseudoCode

Transaction* (Class for object oriented design)

Task 5

```
(Price_At_Day[m][n] , k)
        Profit [][]          ← [k+1] [n]
        Transaction_list ← new Arraylist()
        Team_profit      ← 0

        FOR i = 0 → k + 1
            THEN txn_temp ← new Arraylist()
                FOR j = 1 → n.length()
                  Max1 ← 0
                  Top_txn ← new Transaction(0, 0, 0, 0)
                  FOR x = 0 → j -1
                    THEN FOR stock = 0 → m-1
                              Temp = ((priceAtDay[stock][j] - priceAtDay[stock][x]) +
                                                  Profit[i-1][x])
                              IF max1 < temp
                                THEN max1 = temp
                                top _txn = (stock, x, j, (priceAtDay[stock][j] -
                                                        priceAtDay[stock][x]))
                  IF profit[i][j] < max1
                    THEN txn_temp.add(top_txn)
                  Profit[i][j] = max(max1, profit[i][j-1])
              IF profit[i][n-1] > team_profit
                THEN team_profit = profit[i][n-1]
```

Transaction_list = get_non_overlaps(txn_temp, i)

Println → "Total Profit is:-" + profit[k][n-1]
Return Transaction_list

Time Complexity

The time complexity for the DP tabulation based solution in O($m * n^2 * k$).

Space Complexity

Since two auxiliary of size m*n matrix were used to store the profit transactions. Hence space complexity is O(m*n).

Correctness

Initialization :
- The answer will hold the max profit and its initialized as 0.

Maintenance:
- We buy on day x and sell on day j, total profit made will profit made by this transaction + profit made with k-1 transactions till buy day x.
- Created a new transaction with j as selling day, x as buying day.
- If profit gained till (j-1)th day with i transactions is less than maximum profit we received, then we add that transaction to our list.at each point maximum proft so far will be maximum of profit till (j-1)th day and maximum profit by selling on jth day.
- Updating the temp_profit if profit gained by i transactions on last day is greater in order to achieve maximum profit gained by k transactions in n days.

Termination:
- The helper_functions terminate whenever the 2D array is recursively iterated and the profit[k][n-1] is printed.

Task 6a

Algorithm

The concept used for question 2 is based on a recursive-based memoization algorithm. We are following the steps in particular order:-
- We are using two DP arrays to store temp_profit for both buying and selling a stock m on nth day at kth transaction.

- Further we are initializing two additional DP 3D arrays to memo the store buy and sell transactions.
- We maintain a tuple of transactions with max_profit and pass it in helper_function in addition to the boolean variable sell which is initialized as false.
- Helper_function returns the total_max profit and transaction.

Recurrence Relation

i=0 ->m
k=0 ->n

OPT(i,k) = $taskHelper6aDP(i\,, Bool\,,\ k\,,\ m)$!=0 {
$taskHelper6aDP(i\ +\ 1\,, Bool\,,\ k\,,\ m)$
$taskHelper6aDP(i\,, Bool\,,\ k\ \ -\ 1\,,\ m)$ }

PseudoCode

TransactionL* (Class for object oriented design)

Task 6a

```
(Price_At_Day[m][n] , k)
        Dp_buy [m+1][k+1][n+1]       ← -1
        Dp_sell [m+1][k+1][n+1]      ← -1
        Dp_buy_txn [m+1][k+1][n+1]   ← new TransactionL
        Dp_sell_txn [m+1][k+1][n+1]  ← new TransactionL
        priceAtDay6a = priceAtDay
        TransactionTuple res_tuple = task6a_helper()
        Println → "Max profit is:-" + res_profit.getProfit()
        IF res_tuple.getTransactionList() == null    OR
→           res_tuple.getTransactionList().getTransactionList_L().isEmpty()
          THEN Println → "No Transactions are found to print, profit must be 0 and
                         hence no transaction list is empty"
        Return res_tuple.getTransactionList().getTransactionList_L()
```

Task6a_Helper

```
(int i, boolean sell, int k, int m)

Len_stock ← priceAtDay6a.length()
N ← priceAtDay6a[0].length()
IF (sell)
  THEN IF dp_sell[m][k][i] != -1
           THEN return (dp_sell[m][k][i], dp_sell_txn[m][k][i])
ELSE (sell)
  THEN IF dp_buy[m][k][i] != -1
           THEN return (dp_buy[m][k][i], dp_buy_txn[m][k][i])
IF k == 0
  THEN Return (0, newTransactionL())
IF i == n
  THEN Return (0, newTransactionL())
IF (sell)
   THEN
    Tuple1 = Task6a_helper(i+1, true, k, m )
    Tuple2 = Task6a_helper(i, true, k-1, m )
    Val1 ← Tuple1.getProfit()
    Val2 ← Tuple2.getProfit()
    X1 ← tuple1.getTransactionList()
    X2 ← tuple2.getTransactionList()
    Dp_sell[m][k][i+1] = val1
    Dp_sell_txn[m][k][i+1] = x1.getTransactionList_L()
    Dp_buy[m][k-1][i] = val2
    Dp_buy_txn[m][k-1][i] = x2.getTransactionList_L()
    IF val1 > val2 + priceAtDay6a[m][i]
       THEN return (val1, x1.add(new Transaction(-m, -1, i, priceAtDay6a[m][i]))
    RETURN (val2 + priceAtDay6a[m][i] , x1.add(new Transaction(-m, -1, i,
                  priceAtDay6a[m][i]))
ELSE
       Total-max ← 0
       List x3 ← new Arraylist()
       FOR m1 = 0 ← len_stocks
            Tuple1 = Task6a_helper(i+1, false, k, m1 )
            Tuple2 = Task6a_helper(i+1, true, k, m1 )
            Val1 ← Tuple1.getProfit()
            Val2 ← Tuple2.getProfit()
```

```
            X1 ← tuple1.getTransactionList()
            X2 ← tuple2.getTransactionList()
            Dp_sell[m1][k][i+1] = val2
            Dp_sell_txn[m1][k][i+1] = x2.getTransactionList_L()
            Dp_buy[m1][k][i+1] = val1
            Dp_buy_txn[m1][k][i+1] = x1.getTransactionList_L()
            Val2 ← val2 - priceAtDay6a[mi][i]
            List xf ← new Arraylist()
            List xf2 ← new Arraylist()
            FOR t: x2.getTransactionList_L()
                IF t.getBuyDay == -1
                  THEN xf2.add(new Transaction(t.getStockNumber, i, t.getSellDay,
                                                    t.getProfit - priceAtDay6a[m][i]))
                ELSE xf.add(t)
            Temp ← Math.max(val1, val2)
            IF total_max <  temp
              THEN IF val1 == temp
                        THEN x3 = x1.getTransactionList_L()
                      ELSE xf.addAll(xf2.sublist(0, 1))
                             X3← Arraylist(xf)
                      Total_max = temp
    Return tuple(total_max, TransactionL(x3))
```

Time Complexity

The time complexity for the DP Memoization-based solution in O(m*n*k).

Space Complexity

Since two auxiliary of size c((m+1)*(n+1)*(k+1)) matrix were used to store the largest profit. Hence space complexity is O((m+1)*(n+1)*(k+1)).

Correctness

Initialization :

- The answer will hold the max profit which is stored as a tuple in res_tuple.
- Res_tuple also returns the list of transaction to attain max_profit.

Maintenance:

-

Termination:

- The helper_functions terminate whenever the 2D array is recursively iterated and the max_transaction is returned.
- In helper_function all the edge cases are implemented to increase the time efficiency of algorithm.

Task 6b

Algorithm

The concept used for question 2 is based on a tabulation-based DP algorithm. We are following the steps in particular order:-

- First maintaining profit DP 2D array for k transaction in n days to store max_profit.
- Looping through all the transactions to save the best buy_day for each selling day.
- Nested looping through the price_at_day for each stock on each day to update the best buy_day for y.
- Keeping track of stock that gives us max_profit.
- Appending the transaction array if the (j-1) sell_day achieves more profit than jth day.
- Sorting the temp_profit in descending order of i transaction and calling only the max k transaction that do_not_overlap.
- Returning the list with non-overlapping k transactions.

Bottom-up Tabulation Relation

OPT(i,j) = $task6b\_\ DP(i\ ,\ j) \longrightarrow Max\{DP(i\ ,\ j\ -\ 1)\ ,\ profit(j)\}$

The recurrence relation is used to determine the profit of ith transaction by calculating the maximum profit of (j-1)th and jth.

PseudoCode

Transaction* (Class for object-oriented design)

Task 6b

```
(Price_At_Day[m][n] , k)

        Profit [][] ← k+1 , n+1
        Transaction_list ← new Arraylist
        Team_profit ← 0
        FOR  i = 1 → k + 1
          Prev[] ← [Integer.MIN_VALUE]*m
          LastDayBuy[] ← [0]*m
          Cur_list ← new Arraylist
          FOR  j = 1 → n.length()
             FOR  y = 0 → m.length()
                IF prev[y] < profit[i-1][j-1] - price_At_Day[y][j-1]
                  THEN lastBuyDay[y] = j - 1
                       prev[y] = profit[i-1][j-1] - price_At_Day[y][j-1]
             Prev2← MIN_VALUE
             Mj ← 0
             FOR mi = 0 ← prev.length()
                IF priceAtDay[mi][j] + prev[mi] > prev2
                  THEN prev2 = priceAtDay[mi][j] + prev[mi]
                       Mj = Mi
             IF prev2 > profit[i][j-1]
              THEN cur_list.add (Mj, lastBuyDay[], j, priceAtDay[mj][j] -
                                 priceAtDay[mj][lastBuyDay[mj]] )
             Profit[i][j] = max (profit[i][j-1],  prev2)
         IF profit[i][n-1] > team_profit
          THEN team_profit = profit[i][n-1]
               Transaction_list = get_non_overlaps(cur_list, i)

       Println → "Max profit is:-" + profit[k][n-1]
       Return Transaction_list
```

Time Complexity

The time complexity for the DP tabulation based solution in O(m*n*k).

Space Complexity

Since two auxiliary of size (k+1)*(n+1) matrices were used to store the largest profit. Hence space complexity is O((k+1)*(n+1)).

Correctness

Initialization :

- The answer will hold the max profit[k][n-1] and the transaction list.

Maintenance:

- profit dp to keep track of maximum profit made by k transactions in n days.
- If the maximum profit gained by selling on jth day is greater than profit gained on (j-1)th day in i transaction, then we append the transaction to the list
- We are replacing this list for k times, in the end list of k transactions is used only.
- Once we have transaction with maximum profit, we want to find all other non-overlaping transactions which leads maximum profit in k transaction.

Termination:

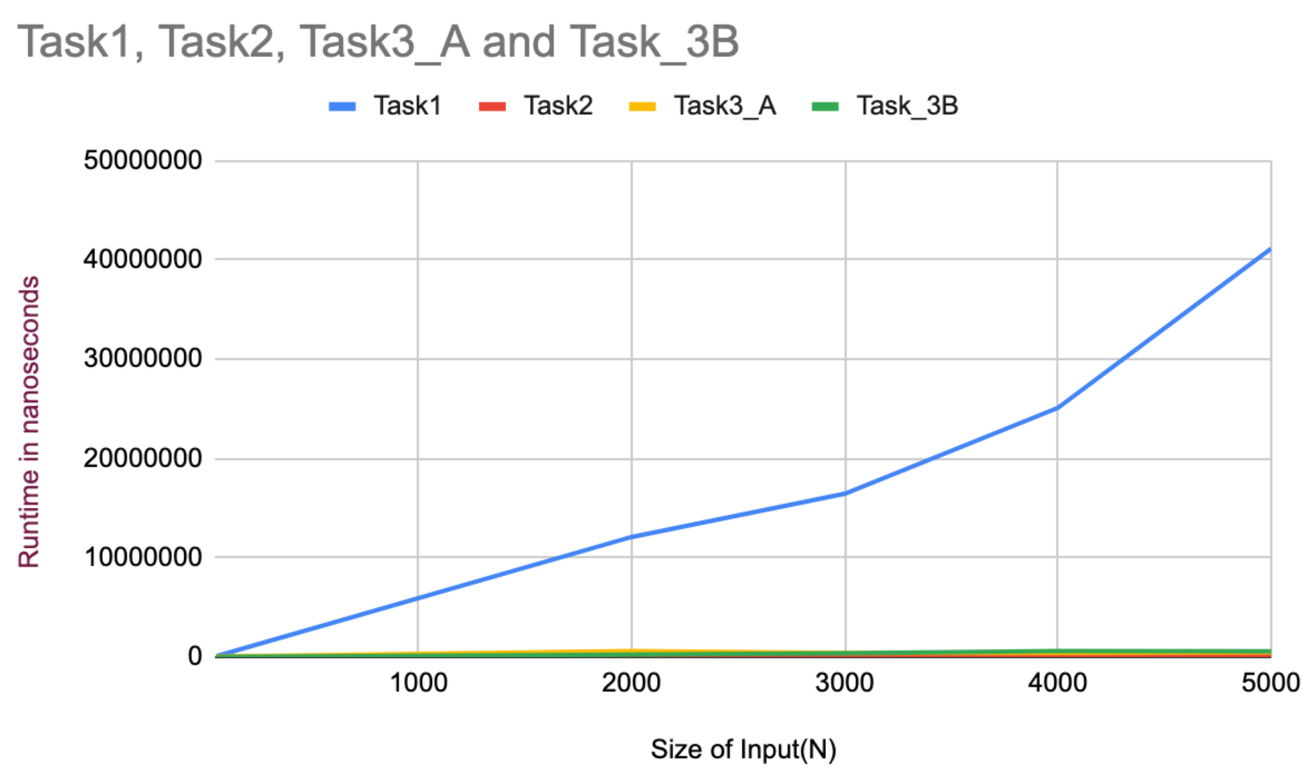
- Once we have the transaction with maximum profit we return the res_tuple.
- Else, if res_tuple is null or isEmpty we return a string stating `“No Transactions are found to print, profit must be 0 and hence no transaction list is empty”`

Experimental Comparative Study

1. PLOT1 - Comparing Task1, Task2, Task_3A, and Task_3B

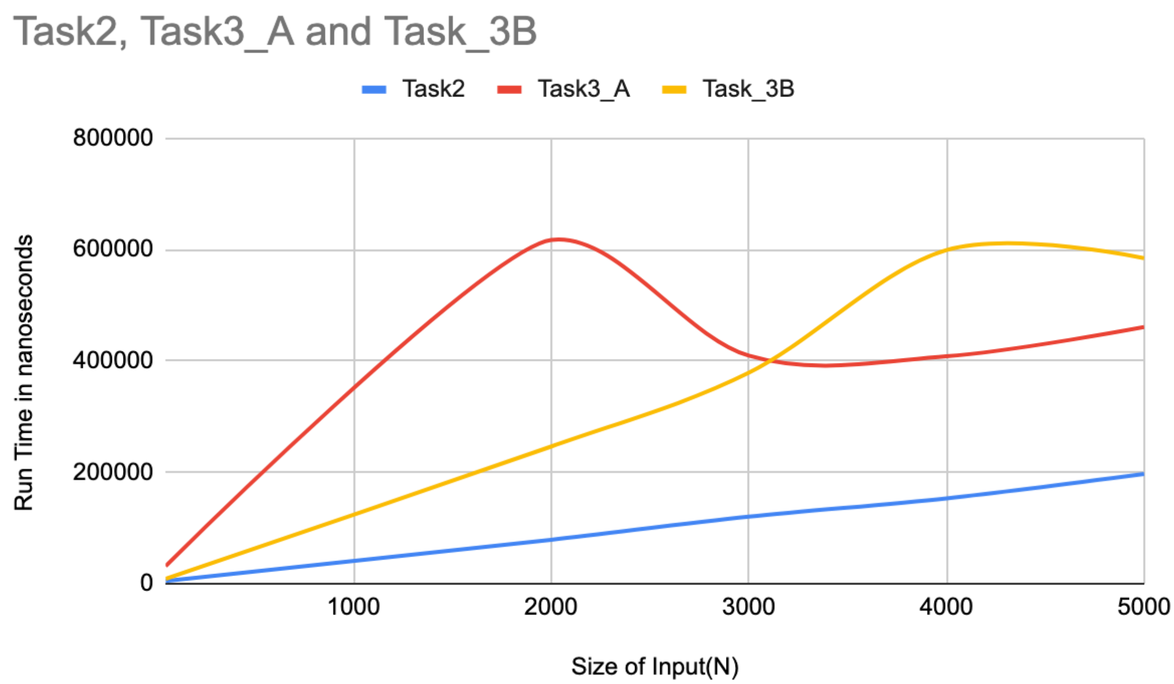
We tested our code on various test cases of different sizes and noted the execution time for each for our comparative study. In this task M is variable and M is fixed.

	Run Time in nanoseconds			
Size of Input(N)	Task1	Task2	Task3_A	Task_3B
50	70375	4875	31500	8541
2000	12093375	78875	617375	246250
3000	16449833	120334	409458	378833
4000	25104042	153250	408417	599084
5000	41135666	197000	460833	584583



- We can see that as the value of n grows, our algorithm for task1 performs much worse as compared to task 2 , task 3a and task3b.

For better comparative analysis we would need to plot more graphs for other algorithms.

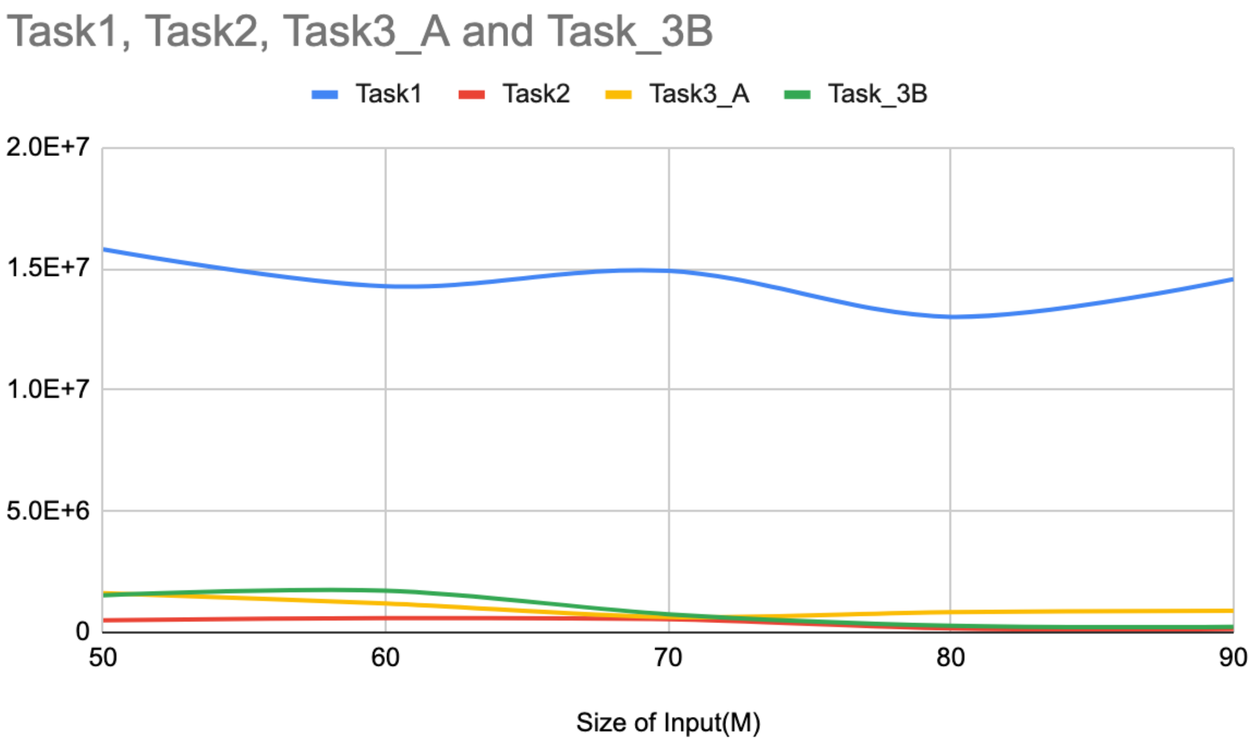


- As the size of input(N) increases firstly task3a performs worse than task2 and task3b but at some point it becomes more efficient than task3b.

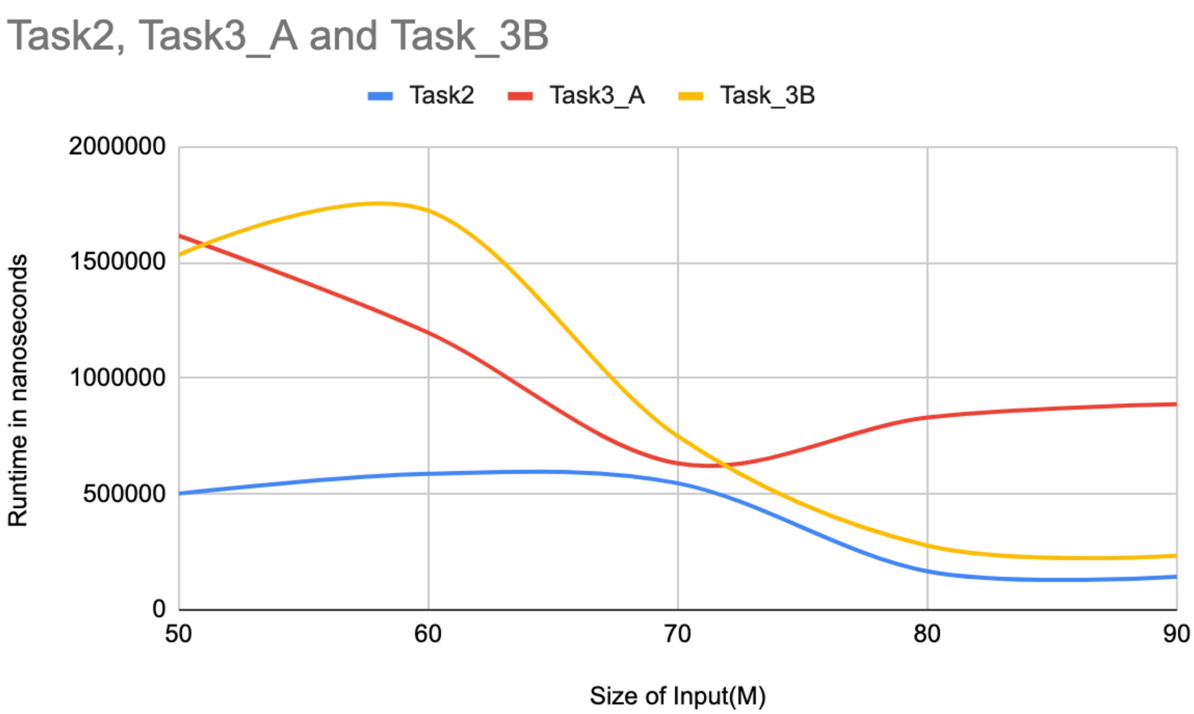
2. PLOT2 - Comparing Task1, Task2, Task_3A, and Task_3B

We tested our code on various test cases of different sizes and noted the execution time for each for our comparative study.In this task M is variable and N is fixed.

Size of Stock(M)	Run Time in nanoseconds			
	Task1	Task2	Task3_A	Task_3B
50	15805667	502000	1616250	1532083
60	14278667	587375	1197042	1723709
70	14915667	546750	632375	750042
80	13017208	166542	830292	277417
90	14573458	142959	887625	233917



- We can see that as the value of M grows, our algorithm for task1 performs much worse as compared to task 2 , task 3a and task3b.

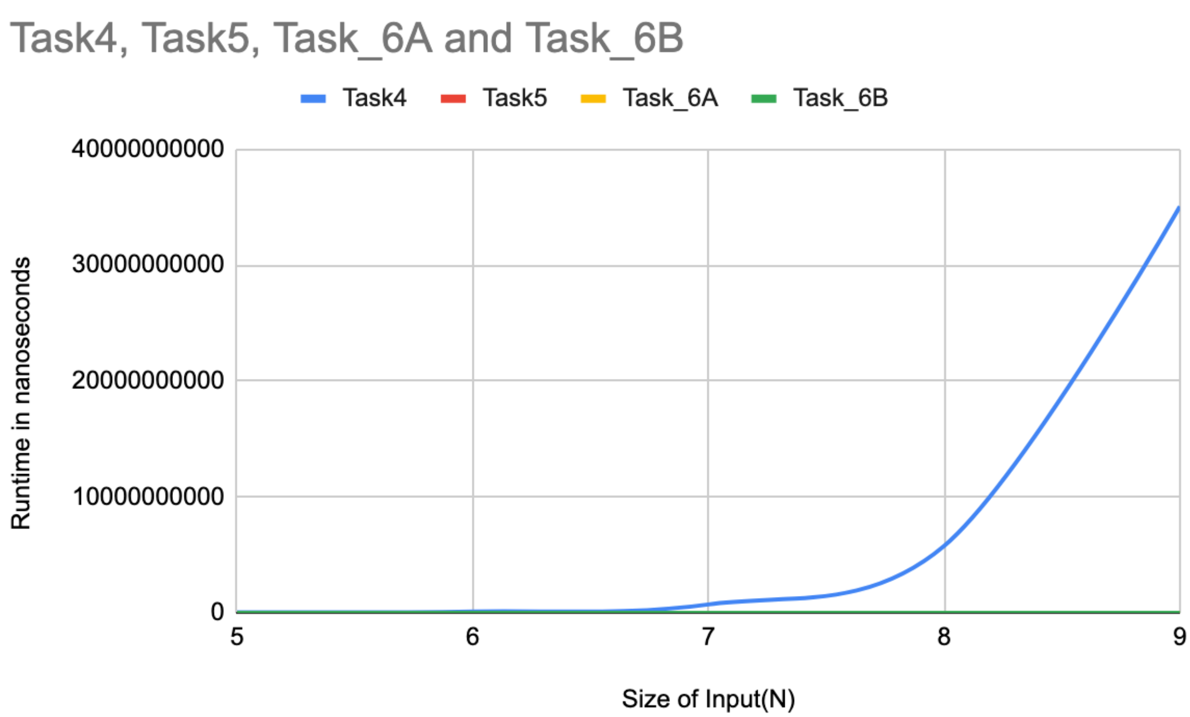


- Task 3a and Task 3b perform worse than Task 2 throughout the variable M.

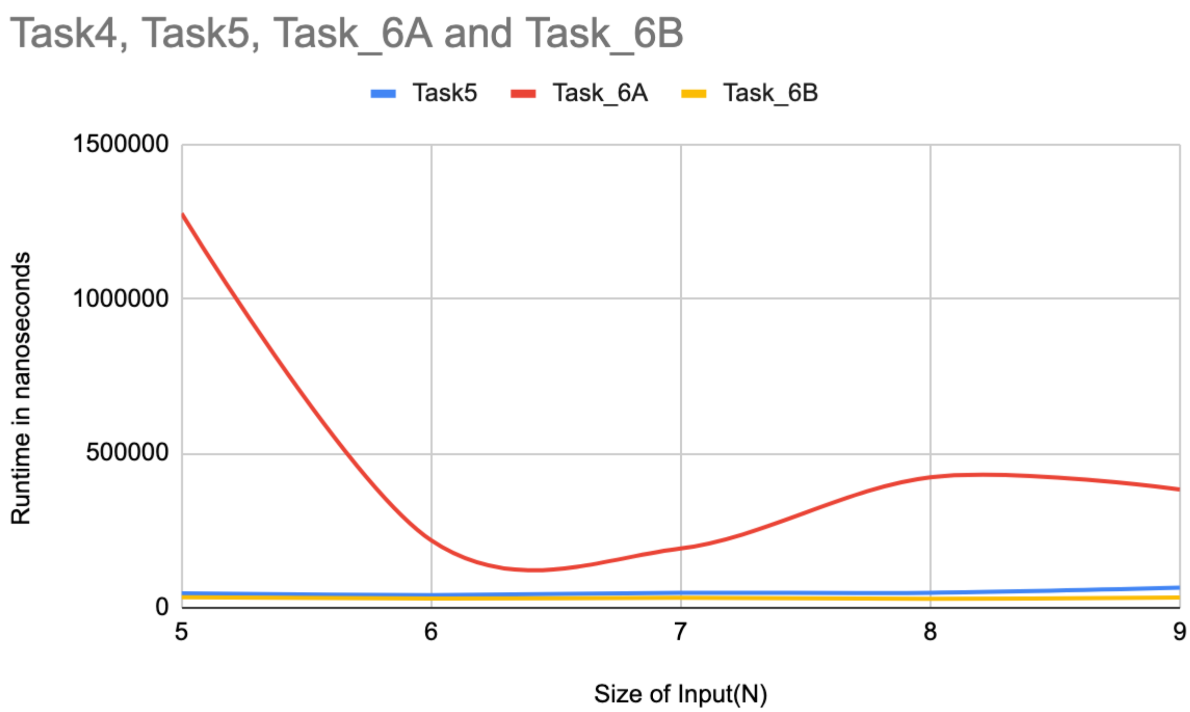
3. PLOT3 - Comparing Task4, Task5, Task_6A, and Task_6B

We tested our code on various test cases of different sizes and noted the execution time for each for our comparative study.In this graph N is variable , M and K are fixed.

	Run Time in nanoseconds			
Size of Input(N)	Task4	Task5	Task_6A	Task_6B
5	43780750	46875	1276292	34209
6	107918000	40834	218333	30042
7	719195208	48292	191916	32291
8	5776317833	48584	422292	29084
9	35066378708	64917	382625	33250



- We can see that as the value of N grows, our algorithm for task4 performs much worse as compared to task 5 , task 6a and task6b.

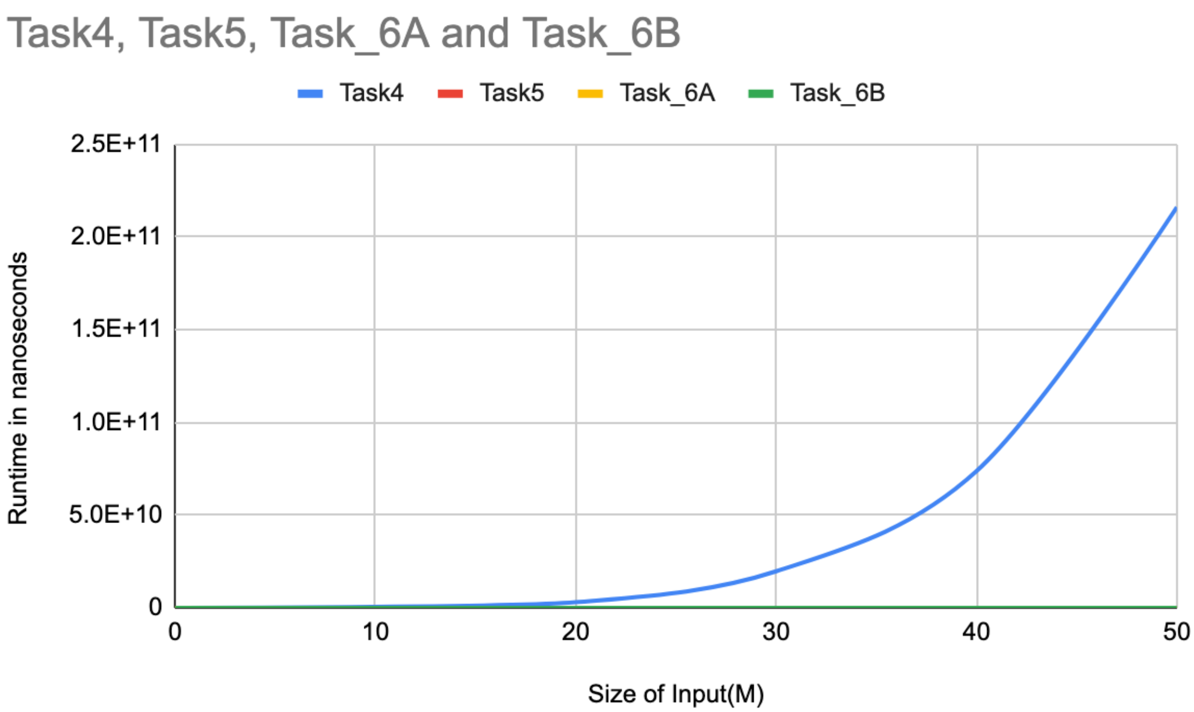


- Task6A is the least runtime efficient in comparison to Task6B and Task 5.

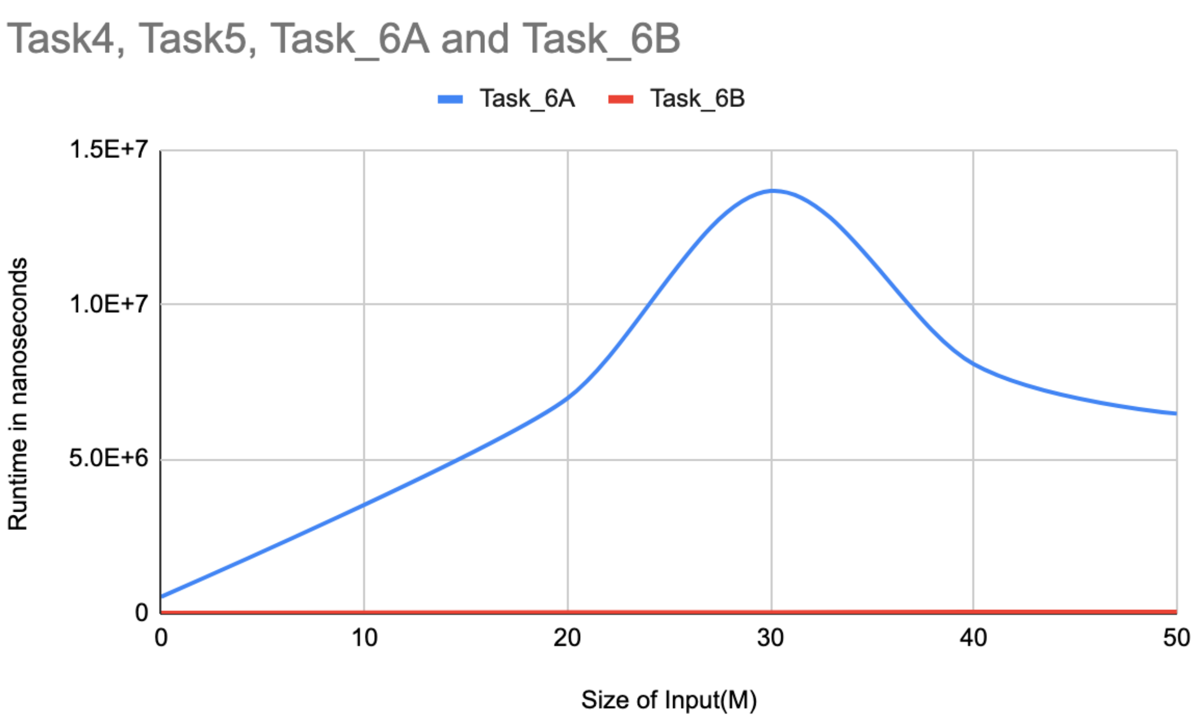
4. PLOT4 - Comparing Task4, Task5, Task_6A, and Task_6B

We tested our code on various test cases of different sizes and noted the execution time for each for our comparative study.In this graph M is variable , N and K are fixed.

Size of Input(M)	Run Time in nanoseconds			
	Task4	Task5	Task_6A	Task_6B
0	4820125	24792	549750	32208
20	2967581583	68167	6966667	44750
30	19693372125	67667	13672834	48750
40	73697679625	114417	8069375	59917
50	216111117625	93250	6470167	61625



- We can see that as the value of M grows, our algorithm for task4 performs much worse as compared to task 5 , task 6a and task6b.It almost goes exponentially worse with the variable M.

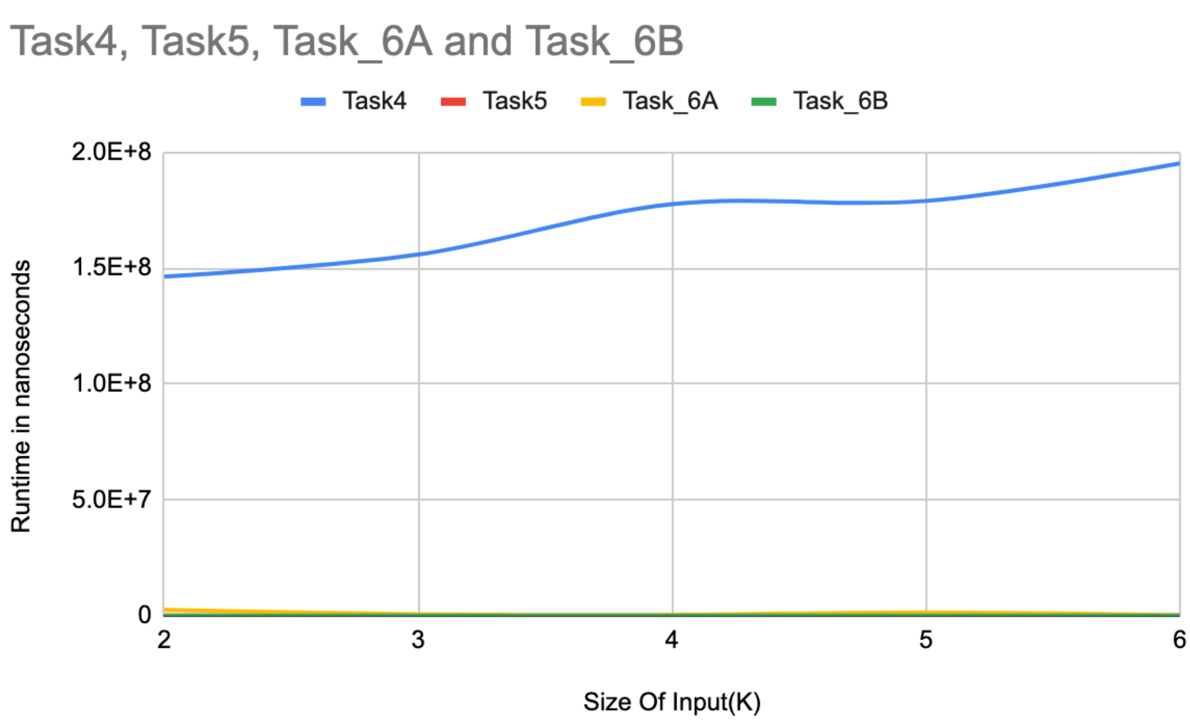


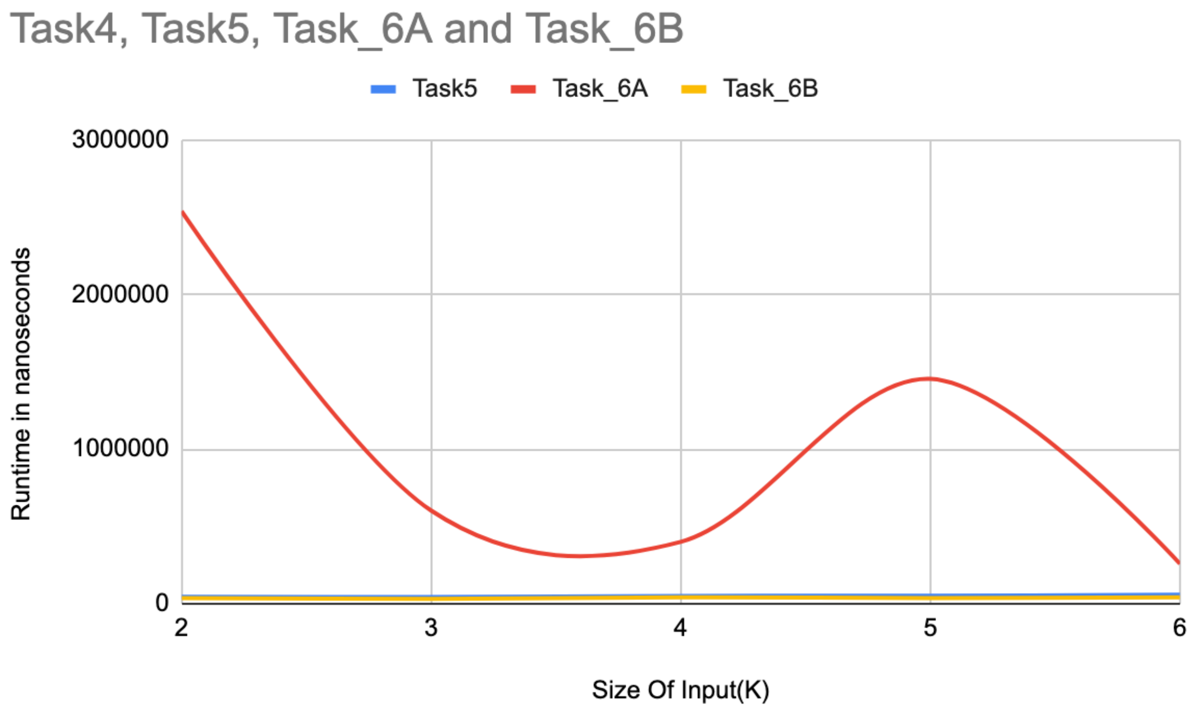
- We can see that tabulation method (bottom up) (task6b) performs better than the memoization (top down approach task 6a) because in top down there are too many recursive call and return statements due to which memory access becomes slower, where as in bottom up method memory access is faster as we directly access previous states from the table.

5. PLOT5 - Comparing Task4, Task5, Task_6A, and Task_6B

We tested our code on various test cases of different sizes and noted the execution time for each for our comparative study.In this graph K is variable , N and M are fixed.

Size Of Input(K)	Run Time in nanoseconds			
	Task4	Task5	Task_6A	Task_6B
2	146397625	44000	2538625	37084
3	155865416	43000	602333	31084
4	177579084	50375	399916	41542
5	179024500	51458	1454625	36958
6	195182125	58042	258208	41416





- We can see Task6A performs better with Size of Input(k) increasing.

Conclusion

Programming Experience:-

This programming project aided us in developing a solution to a problem and prompted us to consider approaches to reduce the problem's time and space complexity. Every programming task had a new learning hidden in it, to find the new learning there were hints between the lines of each task by mentioning the keyword such as “Dynamic programming” “Greedy Approach” and order of time complexity.

Even in DP there were sub-parts including tabulation and memoization and to compare the run-time efficiency of the two.Task 1, 4, 7 were easily achievable as they were brute force approaches, Whereas tasks 3a, 6a, 9a were moderate as they required little optimisation to the brute force approaches. We used memoization to store results of recursion to avoid repetitive calls.

Task 2 was one of it’s own kind as it required us to think greedily about the solution. The implementation part of greedy was pretty straightforward once you know how to choose greedily. Deciding and proving that our greedy solution is always ahead was a little difficult. Task 3b, 5, 8 were moderately hard as they were a step further as compared to recursion and required us to solve the problem using dynamic programming.Task 6b, 9b were the hardest part as we had to think of time optimization on top of dp solutions received from task 5 and task 8.

Task1:- brute force was pretty straight forward. We traverse through each stock one-by-one and find a single transaction that gives us maximum profit. We keep track of maximum so far and return the maximum profit transaction in the end.

Bonus

Task 7

Algorithm

The concept used for question 7 is based on a Brute force recursive-based algorithm. In this we are using a helper_function for recursion which will return a tuple of maximum profit and sequence of.We are comparing the sell_state and the buy_state and both of which consists of 2 cases. Meanwhile adding the cool_down to the tuple2 maintaining the buy_day.
Sell state cases:
1. not sell on current day that means we still hold a stock, so the sell_state for next day will be true. And since we're not doing anything on current day, number of transactions will remain same.
2. by selling on current day, we're finishing 1 transaction, hence k = k-1. And then sell_state for next day will be false as we have to buy before selling again.
Buy state cases:
1.Don't buy on current day, so buy_state for next day will be true and no transactions are made, hence k remains unchanged.
2.Buy on cur day, then sell_state of next day will be true. A transaction is complete when a stock has been sold. Hence k remains unchanged.

Recurrence Relation

i=0 ->n
m=0 ->n

OPT(i,m) = $taskHelperDP(i\,, Bool\ \ , m)$!=0 {
$taskHelperDP(i\ +\ coolDown\ \ +\ 1, Bool\ \ , m)$
$taskHelperDP(i\ +\ 1\,, Bool\,, m)$ }

The recurrence relation in helper function is used to determine the selling day of stock to maximise the profit for each stock and it is being in DP table.

PseudoCode

```
Task7
(Price_at_day [m][n] , cool_down)
priceAtDay 7 → price
Cooldown7 →cool_down
Res_tuple → task7_helper(i:0 , sell: false , m:0)
Return res_tuple

Task7_Helper
  (int i, boolean sell, int m)

  Len_stocks ← priceAtDay7.length()
  N ← priceAtDay7[0].length()
  IF k == 0
    THEN Return (0, newTransactionL())
  IF i == n
    THEN Return (0, newTransactionL())
  IF (sell)
     THEN
       Tuple1 = Task7_helper(i+1, true, m )
       Tuple2 = Task7_helper(i + cool_down7 , False, m )
       Val1 ← Tuple1.getProfit()
       Val2 ← Tuple2.getProfit()
       X1 ← tuple1.getTransactionList()
       X2 ← tuple2.getTransactionList()
```

```
    IF val1 > val2 + priceAtDay7[m][i]
       THEN return (val1, x1.add(new Transaction(-m, -1, i, priceAtDay7[m][i]))
    RETURN (val2 + priceAtDay7[m][i] , x1.add(new Transaction(-m, -1, i,
                    priceAtDay7[m][i]))
ELSE
       Total-max ← 0
       List x3 ← new Arraylist()
       FOR m1 = 0 ← len_stocks
            Tuple1 = Task7_helper(i+1, false, m1 )
            Tuple2 = Task7_helper(i+1, true, m1 )
            Val1 ← Tuple1.getProfit()
            Val2 ← Tuple2.getProfit()
            X1 ← tuple1.getTransactionList()
            X2 ← tuple2.getTransactionList()
            Val2 ← val2 - priceAtDay7a[mi][i]
            List xf ← new Arraylist()
            List xf2 ← new Arraylist()
            FOR t: x2.getTransactionList_L()
                 IF t.getBuyDay == -1
                   THEN xf2.add(new Transaction(t.getStockNumber, i, t.getSellDay,
                                                        t.getProfit - priceAtDay7[m][i]))
                 ELSE xf.add(t)
            Temp ← Math.max(val1, val2)
            IF total_max <  temp
              THEN IF val1 == temp
                         THEN x3 = x1.getTransactionList_L()
                       ELSE xf.addAll(xf2.sublist(0, 1))
                               X3← Arraylist(xf)
                       Total_max = temp
```

Time Complexity

The time complexity for the Brute force recursive based solution in O($m * 2^n$).

Space Complexity

Since two auxiliary of size m*n matrix were used to store the largest profit. Hence space complexity is O(m*n).

Correctness

Initialization :

- The answer will return a res_tuple that is initialized as 0 which will return the set of transactions to calculate max_profit.
- A helper_funtion is being used to fill the res_tuple with buy and sell transactions.

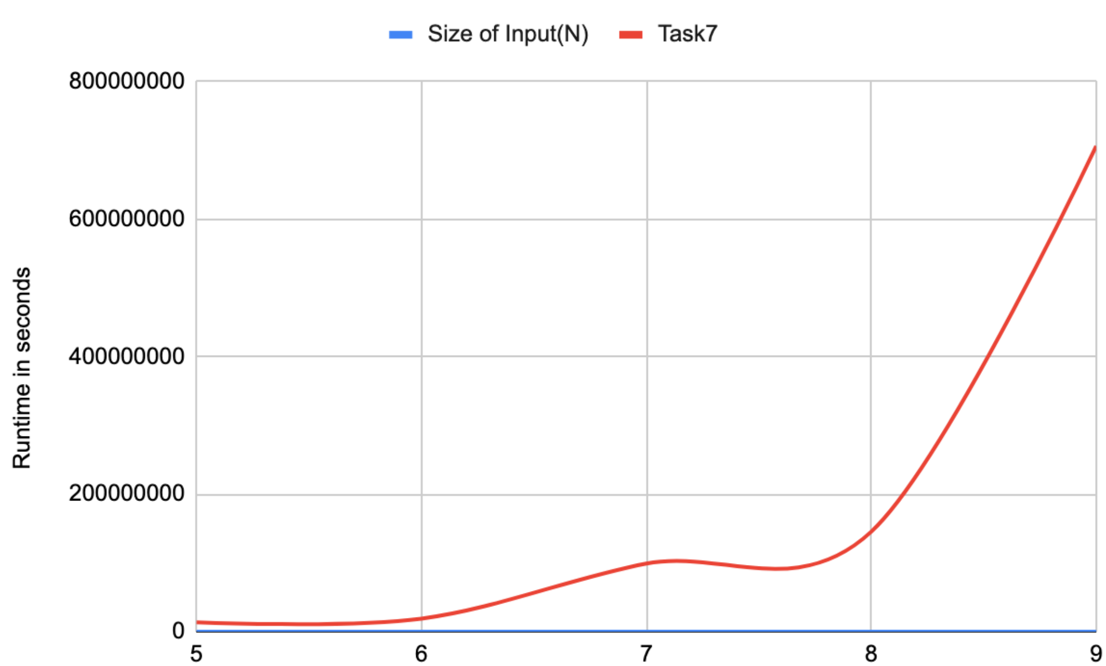
Maintenance:

- Once we have calculated the buy and sell day state and stored it in different tuples.
- We calculate the profit for both the cases and store it in val1 and va2. And then storing the stocks in xf and xf2 which have a buy and and doesn't respectively.
- Meanwhile adding the cool_down to the tuple2 maintaining the buy_day.

Termination:

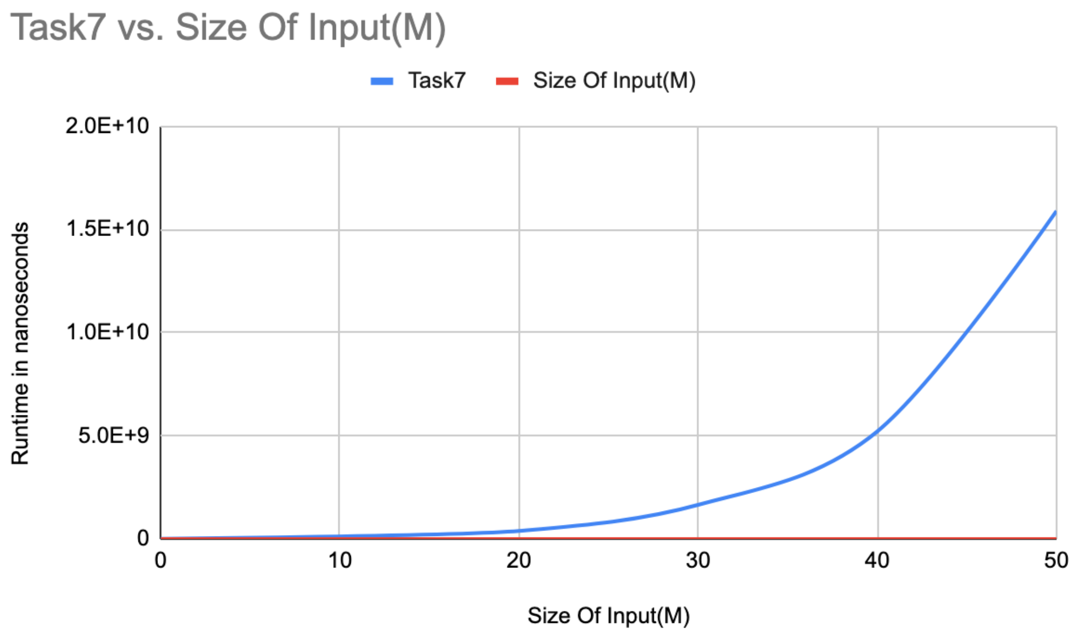
- The helper_functions terminate whenever the 2D array is recursively iterated and the max_transaction is returned.
- In helper_function all the edge cases are implemented to increase the time efficiency of algorithm.

PLOT6 for Algo7 - This is the Runtime analysis of Algo 7 with variable N , and fixed M and Cool_down(c = 3)



	Run Time in nanoseconds
Size of Input(N)	Task7
5	13669834
6	18871125
7	99030625
8	145342459
9	706437333

PLOT7 for Algo7 - This is the Runtime analysis of Algo 7 with variable M , and fixed N and Cool_down(c = 5)



Size Of Input(M)	Run Time in nanoseconds Task7
0	4152666
20	391423916
30	1651277917
40	5212139042
50	15892114250

Task 9b

Algorithm

Recurrence Relation

PseudoCode

Task_9b
(Price_at_day [len_stocks][n] , cool_down)

```
cool[len_stocks][n]          ← [0][0]
sell[len_stocks][n]          ← [0][0]
hold[len_stocks][n]          ← [INT.MIN][INT.MIN]
cool_txn[len_stocks][n]      ← [][]
sell_txn[len_stocks][n]      ← [][]
hold_txn[len_stocks][n]      ← [][]

FOR j=0 → n
  T1 = INT.MIN
  M1← 0
  flagCoolOrSell ← -1
  FOR i =0 → len_stocks
    Cool_val = 0
    Sell_val =0
    IF j==0
      THEN  Cool_val = 0
            Sell_val =0
      ELSE Cool_val = cool[i][j-1]
           Sell_val = sell[i][j-1]
```

Time Complexity

The time complexity for the iterative Bottom -Up based solution in O($m * n$).

Space Complexity

Since two auxiliary of size m*n matrix were used to store the largest profit. Hence space complexity is O(m*n).

Correctness

Initialization :

-

Maintenance:

-

Termination:

-
-