COT5405 - Analysis of Algorithms

Fall 2022

Programming Project Report

Submitted by

Priya Lakshmi Theagarajan (9298-0120)
Anuj Bhardwaj (6964-8276)
Harsha Konda (2014-8357)

On

November 21, 2022.

Instructor
Alper Üngör
Associate Professor, CISE.

Contribution of Team Members

1. Priya Lakshmi Theagarajan (9298-0120)
   Worked on TASK1, TASK2, TASK3A, TASK3B, bonus

2. Anuj Bhardwaj (6964-8276)
   Worked on TASK4, TASK 5

3. Harsha Konda (2014-8357)
   Worked on Report making, TASK 6A, TASK 6B

Problem statement

We are given a array of price predictions for m stocks for n consecutive days. The price of stock i for day j is A[i][j] for i = 1,...,m and j = 1,...,n. You are tasked with finding the maximum possible profit by buying and selling stocks. The predicted price at any day will always be a non-negative integer. You can hold only one share of one stock at a time. You are allowed to buy a stock on the same day you sell another stock. More formally,

Problem1
Given a matrix A of m × n integers (non-negative) representing the predicted prices of m stocks for n days, find a single transaction (buy and sell) that gives maximum profit.

Problem2Given a matrix A of m × n integers (non-negative) representing the predicted prices of m stocks for n days and an integer k (positive), find a sequence of at most k transactions that gives maximum profit. [Hint :- Try to solve for k = 2 first and then expand that solution.]

Design and Analysis of Algorithms

Algorithm 1 - **$\Theta(m * n^2)$** time **brute force** algorithm for solving **Problem1**

DESIGN

Pseudocode of the algorithm is given below, explaining its design.

```
Initialize global variables:
stock = -1,
buyDay = -1,
sellDay = -1

function main()
  Initialization:
  prices = 2D integer list of prices for m stocks and n days

  res = maxProfit(prices)

function findMax(profits)
  max_profit = 0
  FOREACH stock i = 0 to m-1
    max_profit = max(max_profit, profits[i])
    stock = i

function maxProfit(prices)
  profits = Integer list
  mp = Hash map to store buy and sell day of m stocks

  FOREACH stock j = 0 to m-1
    Buy = prices[j][0]; profit = 0; temp_profit = -1;

    FOREACH day i : 0 to n-1
      FOREACH day k : i+1 to n-1
        If (prices[j][k] - prices[j][i])> profit :
          profit = prices[j][k] - prices[j][i]

        If profit > temp_profit:
          temp_profit = profit
```

```
            mp[j] = {i, k} // Buy and sell day of stock j
    add temp_profit to profits list
    max_profit = findMax(profits) // Finds the maximum of profits from m stocks

 // Based on the value of variable 'stock', the corresponding 'buy' and 'sell' day is printed from
HashMap mp

    if stock != -1
        p = mp[stock] // pair of buy and sell day
        buyDay = p.first
        sellDay = p.second

    print stock
    print buyDay
    print sellDay
```

ANALYSIS

Correctness

The first if statement (prices[j][k] - prices[j][i]) > profit = 0 initially ensures profit obtained by selling stock j on day k after buying on day i is always greater than 0 or non-negative. Profit is updated as the k iterates over the innermost for loop for a given buyday i. The for loop over Each buyday i is then implemented over the innermost loop and temp_profit for a given stock j is updated. This does not leave out any possible profitable buyday, sellday pair, satisfying the nature of the brute force. Finally, the function findMax(profits) returns the Maximum profit among all stocks. Hence, the algorithm is correct.

Time Complexity

There are three nested FOREACH loops present in the code. Two nested inner FOREACH loops run over "n" days each.Outer FORECH loop runs over "m" stocks.Therefore, it will result in a time complexity of $\mathbf{\Theta(m * n^2)}$.

Space Complexity

We have a 2D integer list of prices for m stocks and n days occupying O(m*n) space. The other variables constant space, and arrays of size m are also used. So, this problem will have a space complexity of $\mathbf{\Theta(m * n)}$.

Algorithm 2 - **Θ(m ∗ n)** time **greedy** algorithm for solving **Problem1**

DESIGN

Pseudocode of the algorithm is given below, explaining its design.

```
function main()

  Initialization:

  prices = 2D integer list of prices for m stocks and n days

  res = maxProfit(prices)

function maxProfit(prices)

   Initialization:
   stock = -1
   buyDays = Integer list to store the buy part of the transaction
   sellDays = Integer list to store the sell part of the transaction

FOREACH stock j = 0 to m-1
      buy = prices[j][0]; profit = 0; temp_profit = -1;

      FOREACH day i = 1 to n-1

         If prices[j][i] > buy:
            if prices[j][i] - buy > profit:
               profit = prices[j][i] - buy
               sellDays[j] = i

         If prices[j][i] < buy:
            buy = prices[j][i]
            buyDays[j] = i

      If profit > max_profit:
         max_profit = profit
         stock = j

   Print stock
   Print buyDays[stock]
   Print sellDays[stock]
```

ANALYSIS

Correctness

For a given stock j, buy is initialized as first day (day 0) price. For every i iteration from second day (day 1), If price > buy and prices[j][i] - buy > profit = 0 initially, it is taken as sellday = I. Whenever price is greater than buy price and is resulting in a higher profit, sellday and profit are updated ; and if price < buy, whenever price falls below buy, buy and buyday are updated as buy = price and buyday = I. Therefore, invariats are properly maintained to suit the problem requirement thereby establishing the correctness of the greedy algorithm.

Time Complexity

There are two nested FOREACH loops present in the code. The Inner FOREACH loop runs over "n" days. The Outer FOREACH loop runs over "m" stocks. Therefore, it will result in a time complexity of **$\Theta(m * n)$.** It is an improvement over brute force method by a factor of n.

Space Complexity

We have a 2D integer list of prices for m stocks and n days occupying O(m*n) space. Integer lists buyDays and sellDays occupy space O(n), and other variables have constant time space. So, this problem will have a space complexity of **$\Theta(m * n)$**.

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

3A – Memoization

DESIGN

Pseudocode of the algorithm is given below, explaining its design.

```
function main()
   Initialization:
   prices = 2D integer list of prices for m stocks and n days
   buyDays = Integer list of size m
   sellDays = Integer list of size m
   temp_profit = -1
   max_profit = -1

   FOREACH stock i = 0 to m-1
      temp_profit = maxProfit(prices[i], i, buyDays, sellDays)

      if temp_profit > max_profit:
         max_profit = temp_profit
         stock = i

   Print stock
   Print buyDays[stock]
   Print sellDays[stock]

function maxProfit(prices, stockDay, buyDays, sellDays)

   v = 2D Integer list of size m * 2
   return find(prices, 0,1,1,v, stockDay, buyDays, sellDays)

function find(prices, i, k, buy, v, stockDay, buyDays, sellDays)

   if v[i][buy] != -1:
      return v[i][buy]

   if buy:
      q1 = -prices[i] + find(prices, i+1, k, !buy, v, stockDay,buyDays, sellDays)
      q2 = find(prices, i+1, k, buy, v, stockDay,buyDays, sellDays)

      if q1 > q2:
```

```
        buyDays[stockDay] = i

    return v[i][buy] = max(q1, q2)

else:
    s1 = prices[i] + find(prices, i+1, k-1, !buy, v, stockDay, buyDays, sellDays)
    s2 = q2 = find(prices, i+1, k, buy, v, stockDay,buyDays, sellDays)

    if s1 > s2:
        sellDays[stockDay] = i

    return v[i][buy] = max(s1, s2)
```

Time Complexity : O(m * n)
Space Complexity : O(m * n)

Recursive Formula

For buying at i, q1 = -prices[i] + find(prices, i+1, k, !buy, v, stockDay,buyDays, sellDays).

For not buying at i, q2 = find(prices, i+1, k, buy, v, stockDay,buyDays, sellDays).

For selling at i, s1 = prices[i] + find(prices, i+1, k-1, !buy, v, stockDay, buyDays, sellDays).

ANALYSIS

Correctness

For finding buyDay i, q1 where price of i day is deducted and summed up to the return value of find function running recursively from i+1 with no buy constraint. q2 is when no buy takes place on i day and return value of find function for i+1 is given to it. 2D integer list v[i][buy] gets a return value of maximum of q1 and q2, maintaining the variant for selecting buyDay. Similarly, sellDay is also obtained recursively through find function and maximum of s1 and s2 is returned to v[i][buy]. Function maxProfit returns maximum profitable transaction for each stock. A FOREACH loop over m stocks compares maximum profit in case of each stock and returns the final output. Thus, this algorithm is correct.

Time Complexity

FOREACH loop in main function runs over "m" stocks. The find function calls itself recursively over "n" days. So, this problem will have a time complexity of **$\Theta(m * n)$.**

Space Complexity

We have a 2D integer list of prices for m stocks and n days occupying O(m*n) space. Integer lists buyDays and sellDays occupy space O(n), and other variables have constant time space. So, this problem will have a space complexity of **Θ(m ∗ n)**.

3B – BottomUp

DESIGN

Pseudocode of the algorithm is given below, explaining its design.

```
function main()
   Initialization :
   prices = 2D integer list of prices for m stocks and n days
   buyDays = Integer list of size m
   sellDays = Integer list of size m
   temp_profit = -1
   max_profit = -1

   FOREACH stock i = 0 to m-1.
      temp_profit = max_profit(prices[i], buyDays, sellDays, i)

      if temp_profit > max_profit:
         max_profit = temp_profit
         stock = i

   Print stock
   Print buyDays[stock]
   Print sellDays[stock]

function maxProfit(prices, buyDays, sellDays, stockDay)

   Initialization:
   dp = Integer list of size m+1 and all values are set as 0
   min = INTEGER_MAX

   FOREACH stock i = 1 to m
      if prices[i-1] < min:
```

```
     min = prices[i-1]
     buyDays[stockDay] = i-1

    q1 = prices[i-1] - min
    q2 = dp[i-1]

     if q1 > q2:
        sellDays[stockDay] = i-1

    dp[i] = max( prices[i-1]-min, dp[i-1])

  return dp[m].
```

Time Complexity : O(m * n)
Space Complexity : O(m * n)

Iterative Formula

[i] = max (prices[i-1]-min, dp[i-1]) .

ANALYSIS

Correctness

FOREACH stock i, if prices[i-1] < min, then min is updated to prices[i-1] and buyDays[stockDay] to i-1. Else, sellDays is updated to i-1 when prices[i-1] - min > dp[i-1]. Therefore, Max profit in case of each stock is obtained iteratively for each stock. Then a FOREACH loop over m stocks is run to get max_profit and the corresponding stock with its buyDay and sellDay making this transaction. Thus, this algorithm is correct.

Time Complexity

dp[i-1] is used in the calculation of dp[i] iteratively over "n" days. FOREACH loop runs over "m" stocks. So, this problem will have a time complexity of **$\Theta(m * n)$.**

Space Complexity

We have a 2D integer list of prices for m stocks and n days occupying O(m*n) space. Integer lists buyDays and sellDays occupy space O(n), and other variables have constant time space. So, this problem will have a space complexity of **$\Theta(m * n)$**.

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

DESIGN

Pseudocode of the algorithm is given below, explaining its design.

```
Initialize Global Arraylist transactions
function main():
    Initialization:
        int k :   maximum allowed transactions
        int m:    number of stocks
        int n:    number of days
        2D array A[][]:    Input for the prices of stock of size m x n, with all A[i][j]
        representing price of stock I on day n.
    Calc_profits(A,k)
    Ansmap2 = findTransactions(0,k)
    Finaltransactions = ansmap2.get(max)
    Print outputs;

    Calc_profit():
    For all possible start dates:
        For all stocks:
            For all valid sell dates:
                Calculate all possible transactions, i.e. buy and sell operations
                Save to array transactions[]; with start date, end date, stock id,
            profit associated for all transactions
    Return transactions[];

    Findtransactions(int I , int k):
    If K <=0 (transaction not possible) or I < 0 ( transaction[] empty, I is index of
    transaction[i]):
        Don't put anything to the hashmap ansmap();
        Return ansmap;
    If (I = last transaction):
        Add transaction[i] to mylist;
        Put (stock id, mylist) in ansmap;

    Declare a variable next_i ti store the value of next index to check in case we consider a
    transaction;
```

```
For  I < j < size of transaction:
        If ( start date of jth transaction >= end date of ith transaction:
        Next_i = j;

Declare two more hashmaps ansmap2 & ansmap3;
Ansmap2 = Findtransactions( next_i, k-1)  // if we consider we will make transaction[i]
Ansmap3 = Findtransaction(i+1, k); // if we do not consider the ith transaction, we look
at the next transaction to it without skipping any

Ans2 & ans3 are iterators for hasmap ansmap2 & ansmap3 respectively;

If ( profits from ansmap2 > ansmap3):
        Save ansmap2 to ansmap;
Else:
        Save ansmap3 to ansmap;


Return ans map;
```

ANALYSIS

Correctness

Ansmap2 = Findtransactions(next_i, k-1) is calculated if we consider we will make transaction[i]. Ansmap3 = Findtransaction(i+1, k) is calculated if we do not consider the ith transaction, we look at the next transaction to it without skipping any. Therefore, brute force algorithm is correct.

Time Complexity

FOR m stocks, k transactions take place over 2k days. Each day must be checked with every other day in a transaction. So, time complexity is $\Theta(m * n^{2k})$.

Space Complexity

2D array prices of size m x n takes a space of O(m*n).

Algorithm 5 - **$\Theta(m * n^2 * k)$** time **dynamic programming** algorithm for solving **Problem2**

DESIGN

Pseudocode of the algorithm is given below, explaining its design.

```
Initialize Global Arraylist transactions
Initialize a map of map of map called dpmap to store all the states of dp.
function main():
        Initialization:
                int k :   maximum allowed transactions
                int m:   number of stocks
                int n:    number of days
                2D array A[][]:    Input for the prices of stock of size m x n, with all A[i][j]
                representing price of stock I on day n.
        Calc_profits(A,k);
        Declare a hashmap to to store all the states of dp;
        Ansmap2 = findTransactions(0,k);
        Finaltransactions = ansmap2.get(max);
        Print outputs;
        Calc_profit():
        For all possible start dates:
                For all stocks:
                        For all valid sell dates:
                                Calculate all possible transactions, i.e. buy and sell operations
                                Save to array transactions[]; with start date, end date, stock id,
                        profit associated for all transactions
        Return transactions[];

        findTransactions( int I, int k);
                if ( dpmap.get(i) = null):
                        // mapdp map is empty
                        Put I in dpmap;
                If ( dpmap is not empty and dpmap has [i][k]th element:
                        Return the [i][k]th element of dpmap
```

```
        Declare another hashmap called ansmap
        Declare and integer list called mylist
        If K <=0 (transaction not possible) or I < 0 ( transaction[] empty, I is index of
        transaction[i]):
                Put nothing to ansmap;
                if ( dpmap.get(i) = null):
                        // mapdp map is empty
                        Put I in dpmap;
                Get the ith element from dpmap and put it in ansmap;
                Return ansmap;
        If (I = last transaction):
                Add transaction[i] to mylist;
                Put (stock id, mylist) in ansmap;
        Declare a variable next_i ti store the value of next index to check in case we
        consider a transaction;
        For  I < j < size of transaction:
                If ( start date of jth transaction >= end date of ith transaction:
                        Next_i = j;

Declare two more hashmaps ansmap2 & ansmap3;
Ansmap2 = Findtransactions( next_i, k-1)  // if we consider we will make transaction[i]
Ansmap3 = Findtransaction(i+1, k); // if we do not consider the ith transaction, we look
at the next transaction to it without skipping any

Ans2 & ans3 are iterators for hasmap ansmap2 & ansmap3 respectively;

If ( profits from ansmap2 > ansmap3):
        Save ansmap2 to ansmap;
Else:
        Save ansmap3 to ansmap;
```

Recursive Formula

ANALYSIS

Correctness

Time Complexity

$\Theta(m * n^2 * k)$ ***Space Complexi***

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

6A - Memoization

DESIGN

Pseudocode of the algorithm is given below, explaining its design.

```
function main()
   Initialization :
   prices : 2D integer list of prices for m stocks and n days

   maxProfit(m,n,k, prices)

function maxProfit(m, n, k, prices)

   Initialization:
   table : 2D array of size k+1 * n
   set all values of table as -1

   maxDiff : 2D array of size k+1 * m
   set all values of maxDiff as 0

   FOREACH day i: 0 to n.
      table[0][i] = 0

   FOREACH transaction i : 0 to k+1.
      table[i][0] = 0

   FOREACH transaction j : 0 to k+1.
      FOREACH i : 0 to m.
         maxDiff[j][i] = -prices[i][0].

    maximumProfit = calcProfit(m, n-1, k, prices, maxDiff, table)
    dates (Integer array) = calcBuyandSell(m,n,k,prices,table)

   FOREACH i : dates.size() -1 to 0, i = i-3.
      Print dates[i-2] : dates[i-1] : dates[i]
```

```
  return maximumProfit.

 function calcBuyandSell(m,n,k,prices,dp)
   Initialization:
   counter = k
   day = n-1
   list : Integer array

   WHILE(true).
    if counter is 0 or day is 0:
       break.

    if dp[counter][day] equals to dp[counter][day-1]:
       day = day - 1

    else:
       b = false
       FOREACH stock i : 0 to m.
          profit = dp[counter][day] - prices[i][day]
          FOREACH j : day-1 to 0.
             if dp[counter-1][j] - prices[i][j] == profit:
                list.append(i+1)
                list.append(j+1)
                list.append(day+1)
                counter = counter - 1
                day = j
                b = true
                break.
             if b:
                break.

  return list.

 function calcProfit(m, n, k, prices, maxDiff, table)

  if table[k][n] == -1:
     maximum = calcProfit(m, n-1, k, prices, maxDiff, table)
     FOREACH sId : 0 to m.
        maxDiff[k][sId] = max( calcProfit(m, n, k-1, prices, maxDiff, table) -
prices[sId][n],maxDiff[k][sId] )
        maximum = max (prices[sId][n] + maxDiff[k][sId],maximum)
     table[k][n] = maximum
     return maximum.
```

return table[k][n].

Time Complexity : O(m ∗ n ∗ k)
Space Complexity : O(m ∗ n)

Recursive Formula

ANALYSIS

Correctness

Time Complexity

Counter variable is being checked iteratively k times in while loop of function calcBuyandSell. Inside the else condition, two nested FOREACH loops run over m stocks and n days. So, time complexity of the problem is Θ(m ∗ n ∗ k).

Space Complexity

2D integer list of prices for m stocks and n days occupies O(m*n). Default space complexity is O(m*n), assuming k <m,n. 2D array of size k+1 * n and k+1 *m also are there. So, if k > n >m , then Space Complexity is O(k*n) and if k > m >n , it is O(k*m).

6B - BottomUp

DESIGN

Pseudocode of the algorithm is given below, explaining its design.

function main()
    Initialization :
    prices : 2D integer list of prices for m stocks and n days

    maxProfit(prices,n,k,m)

function maxProfit(prices,n,k,m)

    Initialization:

```
dp : 2D array of size k+1 * n+1

FOREACH transaction i: 0 to k+1.
   dp[i][0] = 0

FOREACH day j : 0 to n+1.
   dp[0][j] = 0

FOREACH transaction i : 1 to k+1.
   FOREACH stock v : 0 to m.
      maxDiff = Integer.MIN_VALUE
      FOREACH day j : 1 to n.
         maxDiff = max(dp[i - 1][j - 1] - prices[v][j - 1],maxDiff)
         temp = max(prices[v][j] + maxDiff, dp[i][j - 1])
         dp[i][j] = max(temp, dp[i][j])

dates (Integer array) = calcBuyandSell(m,n,k,prices,dp)

FOREACH i : dates.size() -1 to 0, i = i-3.
   Print dates[i-2] : dates[i-1] : dates[i]

return dp[k][n-1].

function calcBuyandSell(m,n,k,prices,dp)
   Initialization:
   counter = k
   day = n-1
   list : Integer array

   WHILE(true).
      if counter is 0 or day is 0:
         break.

      if dp[counter][day] equals to dp[counter][day-1]:
         day = day - 1

      else:
         b = false
         FOREACH stock i : 0 to m.
            profit = dp[counter][day] - prices[i][day]
            FOREACH j : day-1 to 0.
               if dp[counter-1][j] - prices[i][j] == profit:
                  list.append(i+1)
                  list.append(j+1)
```

```
                list.append(day+1)
                counter = counter - 1
                day = j
                b = true
                break.
            if b:
                break.

    return list.
```

Time Complexity : $O(m * n * k)$
Space Complexity : $O(m * n)$

Iterative Formula

maxDiff = Math.max(dp[i - 1][j - 1] - arr[v][j - 1],maxDiff)

int temp = Math.max(arr[v][j] + maxDiff, dp[i][j - 1])

dp[i][j] = Math.max(temp,dp[i][j])

ANALYSIS

Correctness

Inside maxprofit function, maxDiff = max(dp[i - 1][j - 1] - prices[v][j - 1],maxDiff), temp = max(prices[v][j] + maxDiff, dp[i][j - 1]), dp[i][j] = max(temp, dp[i][j]) were calculated. In function calcBuyandSell, For every stock m, profit = dp[counter][day] - prices[i][day] is checked with dp[counter-1][j] - prices[i][j] before appending and decreasing the counter. Therefore it follows top-down approach and is correct.

Time Complexity

Inside maxProfit function the Outer FOREACH loop runs over k days. Middle FOREACH loop runs over m stocks and Inner FOREACH loop runs over n days resulting in a time complexity of $\Theta(m * n * k)$ for the problem.

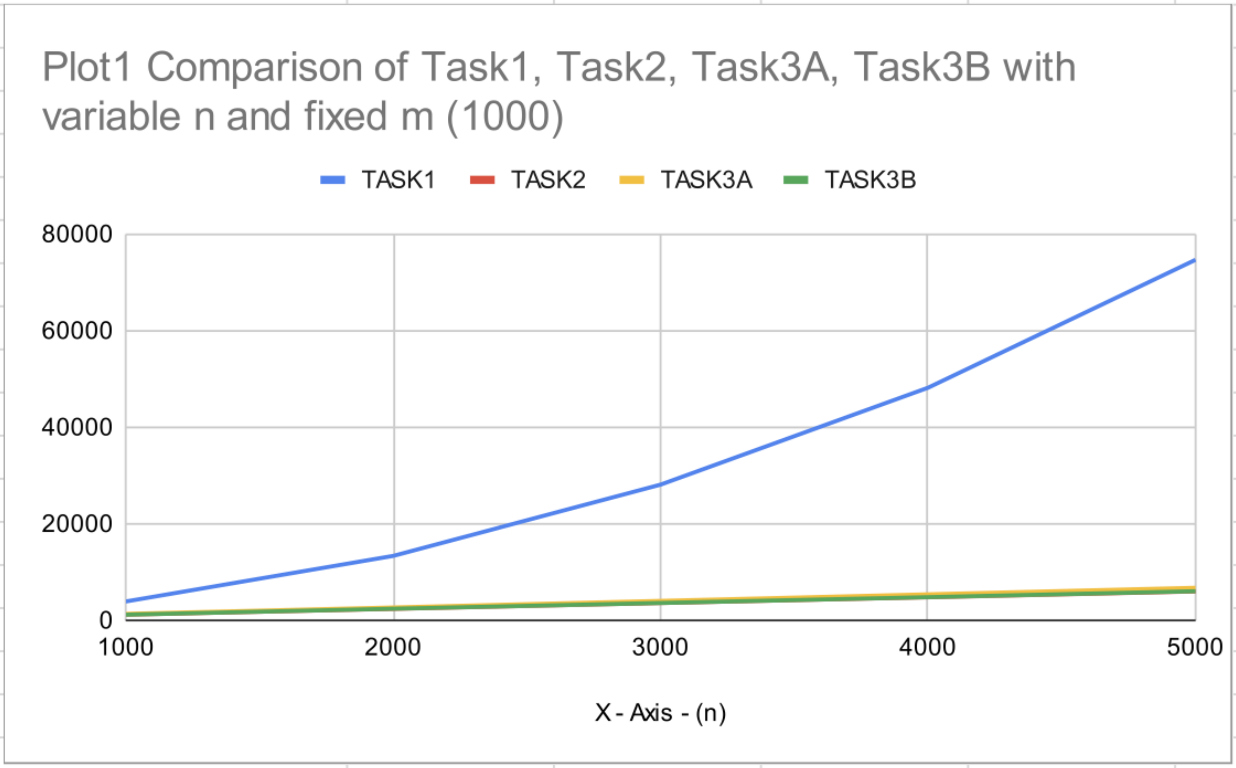
Space Complexity

2D integer list of prices for m stocks and n days in main function. 2D array of size k+1 * n+1 inside maxProfit function. So, the time complexity is O(m*n) if m>k or O(k*n) if k>m.

Experimental Comparative Study

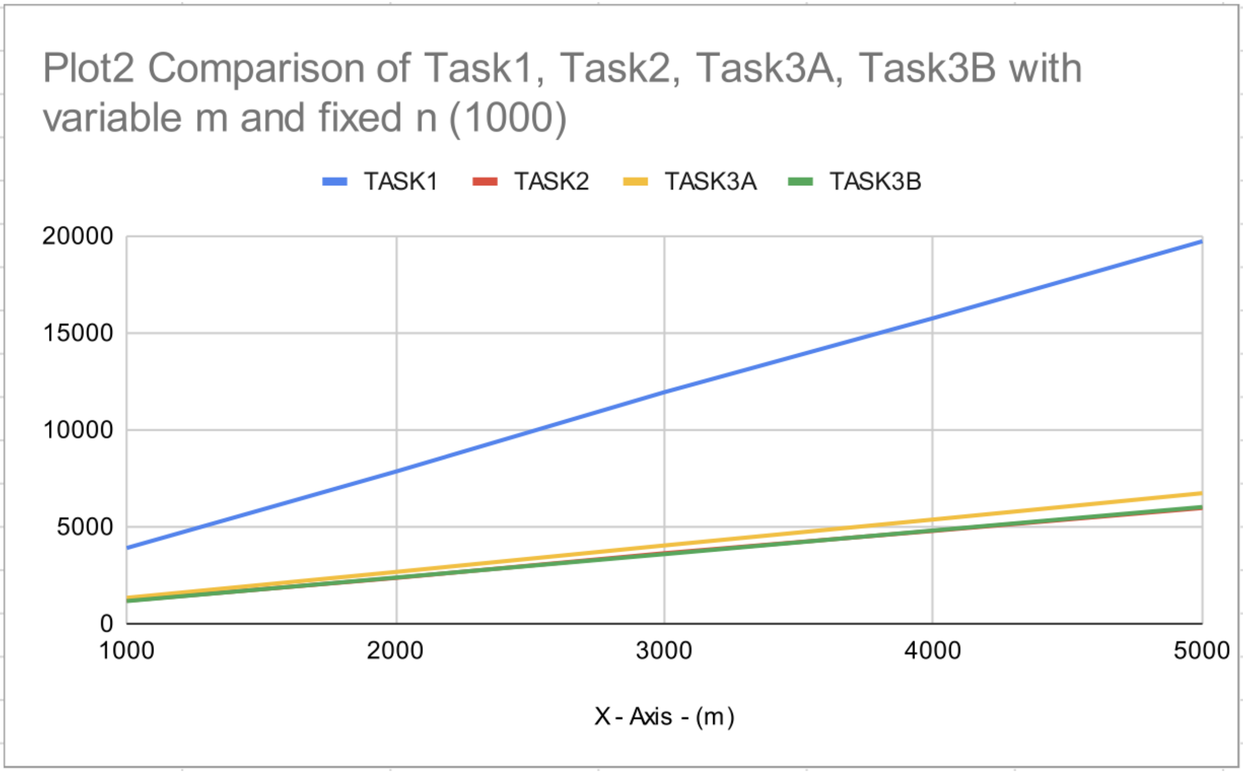
Plot 1 - Comparison of Task1, Task2, Task3A, Task3B with variable n and fixed m

PLOT 1						
			Y-axis (Runtime in milliseconds)			
	X - Axis - (n)	Fixed : m	TASK1	TASK2	TASK3A	TASK3B
	1000	1000	3900	1186	1333	1165
	2000	1000	13342	2375	2662	2399
	3000	1000	28096	3597	4023	3588
	4000	1000	48162	4773	5379	4795
	5000	1000	74654	5987	6736	6024



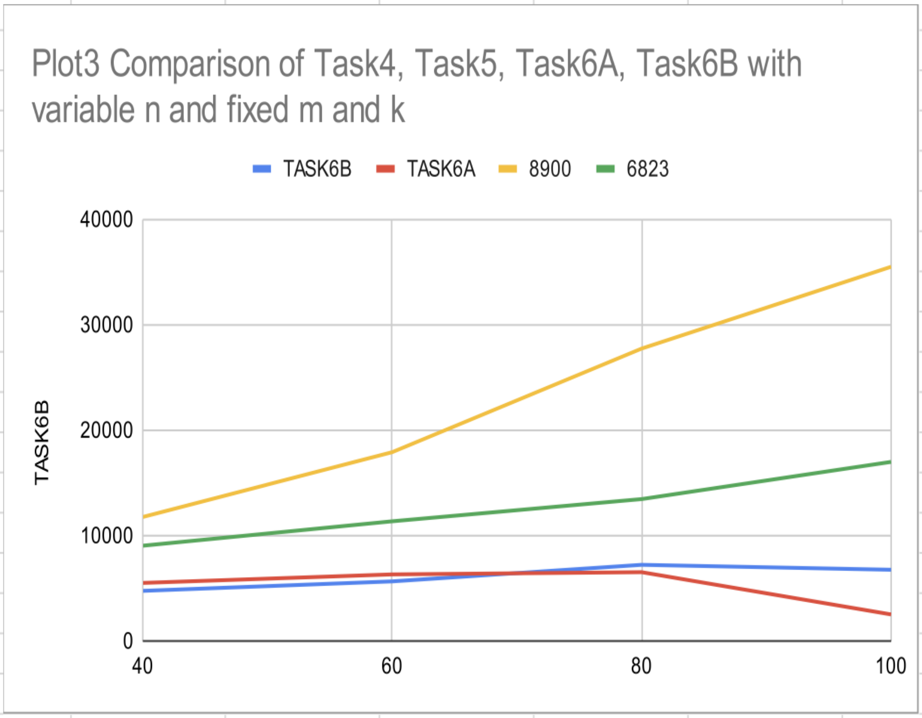
Plot 2 - Comparison of Task1, Task2, Task3A, Task3B with variable m and fixed n

PLOT 2						
			Y-axis (Runtime in milliseconds)			
	X - Axis - (m)	Fixed : n	TASK1	TASK2	TASK3A	TASK3B
	1000	1000	3900	1186	1333	1165
	2000	1000	7840	2358	2666	2377
	3000	1000	11937	3633	4033	3583
	4000	1000	15757	4783	5371	4806
	5000	1000	19718	5965	6728	6017



Plot 3 - Comparison of Task4, Task5, Task6A, Task6B with variable n and fixed m and k

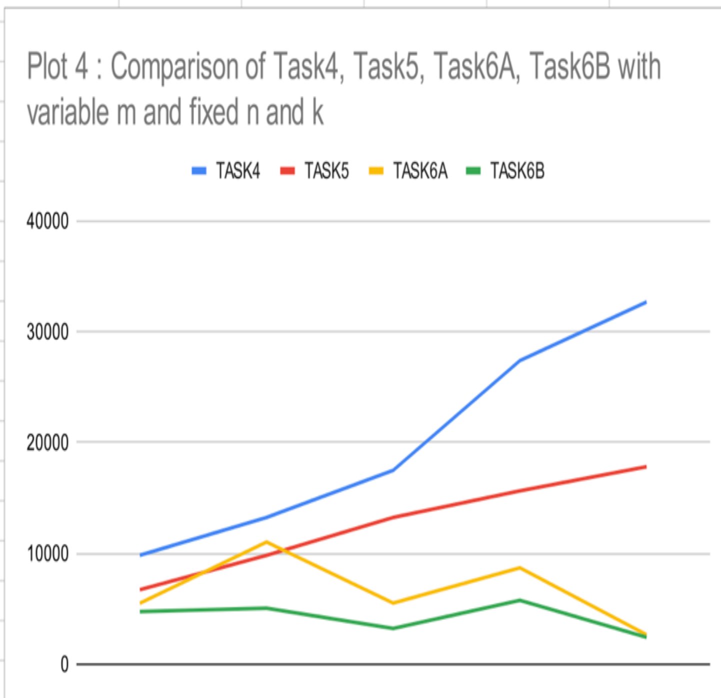
	Y-axis (Runtime in milliseconds)						
X - Axis - (n)		TASK6A	TASK6B	TASK4	TASK5	Fixed : m	K
20		5503	4750	8900	6823	20	2
40		6311	5649	11760	9034	20	2
60		6522	7220	17905.5	11349	20	2
80		2514	6752	27732.32	13456	20	2
100		2696	9372	35492.4	16984	20	2



Plot 4 - Comparison of Task4, Task5, Task6A, Task6B with variable m and fixed n and k

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

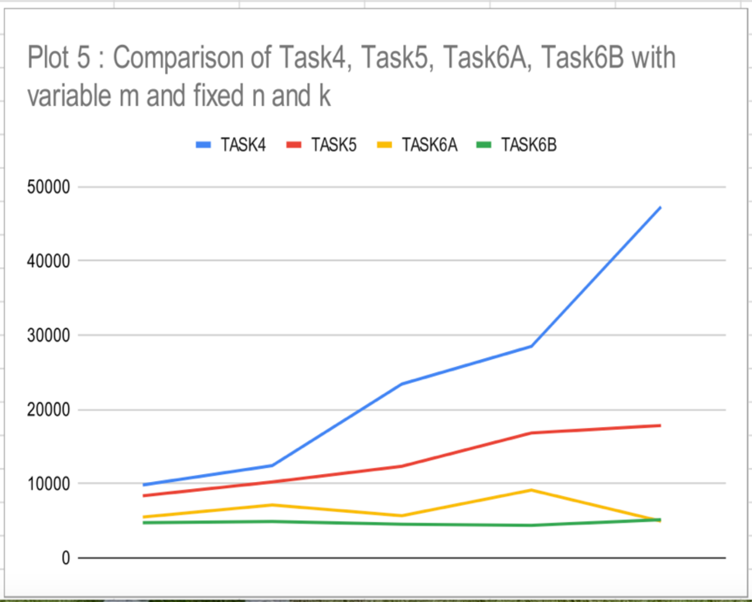
		Y-axis (Runtime in milliseconds)					
X - Axis - (m)		TASK4	TASK5	TASK6A	TASK6B	Fixed : n	K
20		9831	6723	5503	4750	20	2
40		13242	9823	11024	5055	20	2
60		17492	13242	5517	3226	20	2
80		27392	15642	8699	5767	20	2
100		32683	17822	2657	2429	20	2



Plot 5 - Comparison of Task4, Task5, Task6A, Task6B with variable k and fixed m and n

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

	Y-axis (Runtime in milliseconds)					
X - Axis - (k)	TASK4	TASK5	TASK6A	TASK6B	Fixed : n	m
2	9832	8372	5503	4750	20	
3	12434	10233	7132	4907	20	
4	23422	12342	5685	4535	20	
5	28492	16832	9137	4389	20	
6	47293	17832	4947	5144	20	



Conclusion

(on learning experience, ease of implementation, potential technical challenges for each task)

Algorithm1 is easy to implement and straightforward. We have considered all possible transactions and taken the best one as the solution. This algorithm is clearly not optimal, though it is correct. As shown in Plot 1 and Plot 2 in experimental comparative study, execution time increases in O (m * n^2) for brute force approach.

Algorithm2 is an improvement over brute force in terms of time complexity. But, the space complexity remains the same as O(m * n). As in greedy, we find local minimum and maximum price over each pass for a given array. This implementation is also easier as it contains only two FOREACH loops. As shown in Plot 2 in experimental comparative study, execution time is reduced to O (m * n) from O (m * n^2).

If we look at Algorithm3A and Algorithm3B, we observe bottom-up approach is easier to implement than memorization or top-down approach. Plot also indicates running time in case of bottom-up is lower than that in memorization. Running time is O(m*n) in both cases and is almost the same greedy approach.

Algorithm 4 is harder to implement unlike usual brute force algorithms. It took a complexity of O