University of Florida

Department of Computer and Information Science and Engineering

---

Analysis of Algorithms

(COT5405) Fall 2022

Instructor: Alpher Ungor

Programming Project

Name	UFID	Email ID
Venkat Sai Putagumpalla	**3308-8798**	**v.putagumpalla@ufl.edu**
Ratna Prabha Bhairagond	**8827-4983**	**r.bhairagond@ufl.edu**
Priti Shyamrao Gumaste	**4953-5219**	**pgumaste@ufl.edu**

<u>Team Members:</u>

We started working on the project with brainstorming sessions and working out the logics of various approaches for Problem 1, Problem 2 and Problem 3. Then we went ahead and divided the tasks between us so that all of us could single-handedly get to work on the project's various tasks. Ratna and Venkat worked on the dynamic part of Problem 1 and Problem 2 and the bonus part. Priti worked on the brute force of Problem 1 & 2. After our individual tasks were done we discussed our results together and worked on doing the experimentative comparative study.

1. **Ratna Prabha Bhairagond (UFID: 8827-4983)**
2. **Venkat Sai Putagumpalla (UFID: 3308-8798)**
3. **Priti Shyamrao Gumaste (UFID: 4953-5219)**

<u>Problem 1</u> :

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

<u>Problem 2</u> :

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

<u>Problem 3</u> :

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), find the maximum profit with no restriction on the 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.

<u>Algorithm 1</u> : Design a Θ(m * n^2) time brute force algorithm for solving Problem1.

Design/Algorithm:

1. For the brute force approach, we initially start with comparing all the elements of the matrix with each other (prices[k][j]>prices[k][i]) by running a nested loop..
2. If there is an element which is greater than the last element, we take the difference between them and keep updating our maximumProfit if the difference is greater. Here, j = i+1 where i is the current day and j points to the next day value.

```
if(prices[k][j]>prices[k][i])
{           int diff = prices[k][j]-prices[k][i];
             if(diff>maximumStock)
                   maximumStock = diff;     }
```

Correctness of Algorithm:

1. Proof by using Loop invariant, as the algorithm that we have used here is an iterative algorithm.
2. We prove the correctness in three phases, that is the initialization, maintenance and the termination phase.
3. In the initialization part, we start by looking at all the profits after buying and selling the stocks and consider the first difference to be our maximum profit.
4. To maintain the state, we perform the check to track the profits that we could get if we sold a certain stock at jth day after buying it at ith day, i.e, if profit[j] > profit[i] then we update the maximumProfit with the difference diff = prices[k][j]-prices[k][i] and maximumStock = diff; and consider that amount to be our profit.
5. This check is performed for all the elements and is terminated and the old value is maintained if no maximum profit is found. And the condition is true when termination as well. Hence, we proved using the proof of invariant.

Time complexity:
The for loop iterates for all the elements in our matrix for n(n-1)/2 times making it O(n^2) and as we have m stocks, the time complexity is O(m * n^2).

Space complexity:
The space complexity is going to be O(1) , as we have used variables to store and calculate the profits.

<u>Algorithm 2</u> : Design a Θ(m ∗ n) time greedy algorithm for solving Problem1

Design/Algorithm:

1. We are given price of m stocks for n days in m*n matrix and also only one transaction is allowed.
2. We can solve this problem by using a slightly modified version of kadane's algorithm. In kadane's algorithm for finding maximum subarray we end up finding maximum subarray ending at every index of an array.
3. We need to find maximum profit that can be achieved by buying a stock at index i and selling it at index j where i<=j and j goes from 1 to n days (for all n days).
4. Initialize the minPrice = first term, maxProfit = 0 and and then we keep traversing the array.
5. If we find a stock price on a day that is larger than min then we find difference between the stock price and min and if the difference greater than maxProfit we update maxProfit and if stock price is less than minPrice we update the minPrice with current stock price

   minPrice=min{minPrice,stockPrice[i]}
   maxProfit=max{maxProfit,stockPrice[i]-minPrice}

Correctness of Algorithm:
Since the number of transactions possible are only one we can rethink the problem 1 as finding max difference[A[j]-A[i]] between two index in an array A where i<j, In order to find the max difference at any index j we should keep track of least value that appears before that index, in above algorithm we are doing exactly that keeping track of least(min) value that appears before the current index

Time Complexity:
Above algorithm takes Θ(n) time because we are doing only pass through the array. We can extend it to M stocks by finding maxProfit for each stock and finding maximum out of maxProfit for each stock.

globalMaxProfit=max($\forall$ j $\in$ {1,…,M}{maxProfit[j]})

Since we are Passing through M stocks for n days the Time Complexity would be Θ(m*n).

Space Complexity:
Space complexity of this algorithm is O(1) as we are just variables to store the result.

<u>Algorithm 3</u> : Design a Θ(m ∗ n) time dynamic programming algorithm for solving Problem1.

Design/Algorithm:

Base cases
OPT(i, 1) = 0, for day 1
OPT(0, j) = 0, for stock 0

OPT(i, j) = max { prices[i][j] - prices[i][P[i][j]], OPT(i - 1, j), OPT(i, j - 1) },
where P[x][y] is the day of the least price of the stock x before day y.

Solution: OPT(m, n)

1. We are given price of m stocks for n days in m*n matrix and the number of transactions allowed is 1.
2. We consider a P[m][n] matrix where P[i][j] is the day of the least price of the stock i before day j.
3. Base case, for every i from 1 to m, P[i][1] = 1. The least price of stock before day 1 would be day 1 itself.
4. We loop from i=1 to m and the nested loop from j=1 to n to calculate each P[i][j].
5. If (prices[i][P[i][j - 1]]) < (prices[i][j - 1])
   then P[i][j] = (P[i][j - 1])
   else P[i][j] = (j - 1)
6. We consider a M[m][n] matrix which will store the optimal result (max profit) M[i][j], for i stocks and j days.
7. Base case, for every i=0 to m, M[i][1] = 0, as the profit would be 0 for day 1.
8. Base case, for every j=0 to n, M[0][j] = 0, as the profit would be 0 for 0 stocks.
9. For calculating M[i][j] for every i from 1 to m and j from 1 to n, we use two nested for loops. Loop from i=1 to m and nested loop from j=1 to n.
10. M[i][j] = max {prices[i][j] - prices[i][P[i][j]], M[i - 1][j], M[i][j - 1]}
11. M[m][n] will have the resultant maximum profit.
12. To backtrack the solution, we traverse through M from M[m][n], to get the stock number, buy day and sell day.

Correctness of Algorithm:

1. We will prove this by loop invariant.
2. Let M[i][j] be the maximum profit terminating at stock i for 1 ≤ i ≤ m and day j for 1 ≤ j ≤ n.

3. For Initialisation, for zero stocks the maximum profit will be zero, i.e., M[0][j] = 0, for all $1 \leq j \leq n$ and, for first day, as only one stock could be bought and no stock can be sold, the maximum profit will be zero, i.e., M[i][1] = 0, for all $1 \leq i \leq m$.
4. For maintenance, consider M[i][j] for some $0 \leq i \leq m$ and $0 \leq j \leq n$. We denote the day of the lowest stock price before day j as P[i][j] and M[i-1][j] as maximum profit terminating at stock i-1 and M[i][j-1] as maximum profit terminating at day j-1. If prices[i][j] is the maximum price of stock I till now, the maximum profit becomes prices[i][j] – prices[i][P[i][j]], i.e., the difference between the maximum stock price and minimum stock price. If that is not the case, the maximum stock price would be before day j, therefore we consider M[i][j-1] or else, the maximum profit might be of a stock before i, then we consider M[i-1][j]. Hence, for every stock i and day j, the problem narrows down to finding a maximum between just three numbers: M[i-1][j], M[i][j-1] and prices[i][j] – prices[i][P[i][j]]. It therefore exhibits the optimal substructure and holds the maintenance condition.
5. As the maintenance condition is followed repeatedly, the condition holds true during termination too. Since when we have $i = m + 1$, $j = n + 1$ we will already have the maximum for $i = m$ and $j = n$ as expected.

Time Complexity:

1. To calculate P[i][j], we traverse each element once with the help of a two nested loop and do constant time operation for each iteration. Hence, the time complexity is O(n * m).
2. Similarly, to calculate M[i][j], we traverse each element once with the help of a two nested loop and do constant time operation for each iteration. Hence, the time complexity is O(n * m).
3. For backtracking, to traverse through the elements in M[m][n], it takes <m*n recursive calls, therefore the time complexity is O(m * n).
4. Therefore the overall time complexity comes up to O(m * n).

Space Complexity:

The algorithm uses extra space, i.e matrix to store optimal maximum profit M[m][n] and matrix to store the day of the least stock price before current day P[m][n] and hence the space complexity is O(row.column) = O(m * n).

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

Design/Algorithm:

1. For the brute force approach, we start with traversing through all the elements and finding the maximum possible profit in the transactions after buying and selling the stocks.
2. We need to consider multiple scenarios of when we hold on to the stock, the day we sell the stocks, and if we decide not to sell or buy any stock on some particular day.
3. If we are holding on to a stock we check for the maximum profit that the price would give us and make interactive calls.

   helpInitialFunc resultHelper = findSolution(stockPrices, day + 1, k, considerBuy, ID, boughtDay, (ArrayList<ArrayList<Integer>>)result.clone());

4. For selling any stock, we first calculate the difference between the prices of the day we bought the stock and the present day price

   isDifference = stockPrices[ID][day] - stockPrices[ID][boughtDay].
   We add the calculated difference to the profit
   profit = profit + isDifference;
   if the profit > maximumProfit
   update the maximumProfit value.

5. To buy any stock we make calls to the findsolution() function to iterate through the prices to check for the highest price.
   resultHelper = findSolution(stockPrices, day + 1, k, true, i, day, (ArrayList<ArrayList<Integer>>)result.clone());

6. We store the indices of the transactions that give us the maximum profit while iterating through them in an arraylist and return it in the end.

Correctness of Algorithm:

1. We can prove the correctness of this algorithm using the proof by induction.
2. Considering the base case when k = 1, that is we need to find only 1 transaction including buy and sell indices that gives us the maximum profit.
3. Assuming that we bought a stock at day i and after a certain period of time stockPrices[ID][day] and stockPrices[ID][boughtDay] are the prices of the stocks when we decide to sell the stock with the profit -

maximumProfit = profit + isDifference;
where , isDifference = stockPrices[ID][day] - stockPrices[ID][boughtDay];

4. This would necessarily give us the required maximumProfit. By inductive step, we can say that if that is true for k = 1 transaction that means by proof of induction it is also true for k transactions.
5. By implications, we can prove that this algorithm is current and also works for k transactions.

Time Complexity:

We have nested loops over the prices list for n number of days and m stocks, we iterate over the matrix with O(m * n^2) complexity. To iterate and find the k transactions with selling and buying of stocks that give us the maximum profit we will have the time complexity of O(m* n^ k*2).

Space Complexity

The space complexity would be O(n) as we are using an ArrayList to store the transactions and variables to store the maximum profit.

<u>Algorithm 5</u> : Design a Θ(m *n^2*k) time dynamic programming algorithm for solving Problem2.

Design/Algorithm:

1. From Algorithm 3(DP for 1 transaction) we know that,

   dp[k][day]= max{ dp[k][day-1], max{ (∀m∈{1,…,day-1}(dp[k-1][m]+
   stockPrice[day]-stockPrice[m]) };
   day->currentDay;
   m->day on which we buy stock;
   k->total transactions;

2. If we buy stock at day i and sell at day j (i<j) and we have performed total k number of transactions, then maxProfit if we add the current profit (stockprice[j]-stockprice[i]) and maximum of profit that can obtained by selling any stock at day i with k-1 number of transactions, since we have multiple transactions and multiple stocks to consider prices of one stock influence the maxProfit we can get from other stock till day d so, we also need to check what is the value of s-1 (where s is current stock number) before filling dp array

   dp[k][stockNumber][day]=max {dp[k][s-1][day],dp[k][s][day-1],
   max(∀m∈{1,…,day-1}(dp[k-1][**totalStocks**][m]+stockPrice[s][day]-stockPrice[s][m])
)}

   day->currentDay
   m->day on which we buy stock;
   k->total transactions
   s->Stock number

3. Here we looking back at dp[k-1][totalStocks][m] instead of dp[k-1][s][m] because the maximum profit that can be obtained with any number of transaction will always be present in last row

Correctness of Algorithm:

1. We will prove this algorithm by Induction
2. Assertion: The index dp[k+1][m+1][n] stores max profit that can be obtained by using k number of transactions

3. Base case When k=0, we can say that dp[0][m][n]=0 as there are no transactions allowed the maximum profit that we can get is 0
4. Hypothesis: Assume that Algorithm gives correct output for k number of transactions
5. Inductive step: we need to prove that algorithm works for k+1 number of transaction
6. From above formula We know that
   dp[k][stockNumber][day]=max {dp[k][s-1][day],dp[k][s][day-1],
   max(∀m∈{1,…,day-1}(dp[k-1][totalStocks][m]+stockPrice[s][day]-stockPrice[s][m])
))}
   Now substitute k+1 in above formula

   dp[k+1][stockNumber][day]=max {dp[k+1][s-1][day],dp[k+1][s][day-1],
   max(∀m∈{1,…,day-1}(dp[k][totalStocks][m]+stockPrice[s][day]-stockPrice[s][m]))}

In the above formula if we further expand both dp[k+1][s-1][day], dp[k+1][s][day-1] both of them will go k number of transactions which was assumed as true already so continuing further we know dp[k][totalStocks][m] gives correct output based on induction hypothesis now we are only left with max(∀m∈{1,…,day-1} (stockPrice[s][day]-stockPrice[s][m]))} as we are checking max value for all of the possible days we will get maximum profit from above equation. Therefore the assertion is true as we get maxprofit at index k.

Time Complexity:

For filling a value in dp table we are checking three things

1. What is value of dp table at d-1 day with same stock number at same transactions which takes constant time
2. What is value of dp table at same day with stock number=stocknumber-1 at same transactions also takes constant time
3. And lastly we are checking all possible days that are less than current day to buy a stock which takes d number of iterations
4. By combining above three steps for any one stock s with n number of days and for a single the number iterations it takes is 1+2+..+d+....+n which is equal to n(n+1)/2, number of iterations it takes for k number of transaction is k*n(n+1)/2

So for m number of stocks it takes m*n(n+1)/2*k iterations which can can represented as Θ(m*n^2*k) in time complexity.

Space Complexity:

Space Complexity of the algorithm is O(k*m*n) as we are using a three dimensional array to store the maxprofit that we get various number transactions and stock number and days

<u>Algorithm 6</u> : Design a Θ(m *n *k) time dynamic programming algorithm for solving Problem2.

Design/Algorithm:

1. This Algorithm is almost similar to Algorithm 5. The only thing we are doing differently is instead of checking all possible days that are less than the current day to buy a stock, we introduce a new variable which keeps track of max difference.
2. Previously what we were doing on day d with k transactions and m number of stocks and stock number s is following:

   Stockprice[s][day]-stockprice[s][1]+dp[k-1][m][1]
   Stockprice[s][day]-stockprice[s][2]+dp[k-1][m][2]
   Stockprice[s][day]-stockprice[s][3]+dp[k-1][m][3]
   .
   .
   .
   Stockprice[s][day]-stockprice[s][d-1]+dp[k-1][m][d-1]

3. In above the value Stockprice[s][day] remains constant through all days only stockprice[s][d-1] and dp[k-1][m][d-1] changes with days and also we don't need to calculate all the values up to d-1 because they are already computed in previous iterations so we just need to calculate the value at d-1 and compare it previous max difference and update max difference if d-1 is greater, so the new formula comes as

   dp[k][stockNumber][day]=max {dp[k][s-1][day],dp[k][s][day-1],
   stockPrice[s][day]+maxiumDifference)}

   day->currentDay
   m->day on which we buy stock;
   k->total transactions
   s->Stock number

   Formula for max difference:
   maxiumDifference = max(maxiumDifference,dp[k-1][totalStocks][day]-stocks[s][day])

Correctness of Algorithm:

Here we are just going prove that maxDifference formula works
Let's they are d number of days and stocknumber s and k transactions what we were doing previously.

day=1
Stocks[s][1]-stocks[s][0]+dp[k][s][0]
day=2
Stocks[s][2]-stocks[s][0]+dp[k][s][0]
Stocks[s][2]-stocks[s][1]+dp[k][s][1]
day=3
Stocks[s][2]-stocks[s][0]+dp[k][s][0]
Stocks[s][2]-stocks[s][1]+dp[k][s][1]
Stocks[s][2]-stocks[s][2]+dp[k][s][2]
day=d
Stocks[s][2]-stocks[s][0]+dp[k][s][0]
Stocks[s][2]-stocks[s][1]+dp[k][s][1]
Stocks[s][2]-stocks[s][2]+dp[k][s][2]
.
.
.
Stocks[s][2]-stocks[s][d-1]+dp[k][s][d-1]

In the above cases if we see we repeatedly calculating Stocks[s][1]-stocks[s][0]+dp[k][s][0] d times we can avoid this repetition by just calculating value for d-1 as all values below that are already calculated in day d-1, And also we can keep of track max difference by simply comparing current difference to it

Time Complexity:

Since we eliminated the requirement of checking all possible days d-1 for day d for every step all three steps mentioned in algorithm 5 takes constant time, so for any one stock s with n number of days the number iterations and for a single the number iterations it takes is 1+1+..+1+....+1 which is equal to n, number of iterations it takes to fill k number of transaction is k*n

So for m number of stocks it takes m*n*k iterations which can can represented as $\Theta(m*n*k)$ in time complexity

Space Complexity:

Space Complexity of the algorithm is $O(k*m*n)$ as we are using a three dimensional array to store the maxprofit that we get various number transactions and stock number and days

<u>Algorithm 7:</u> Design a Θ(m *2^n) time brute force algorithm for solving Problem3

Design/Algorithm:

1. Usually when we consider ideal brute force in which we have selecting m stocks to buy in every buy cycle it takes very high complexity
2. In order to bring the algorithm time complexity to m*2^n what we need to do is instead of considering to buy all m stocks in a buy cycle just select the stock buy which have max profit in a given buy day and sell day
3. Also we are using maxprofit variable to keep track of maximum profit that can be obtained until a certain day in order to print transactions at the end

Correctness of Algorithm:
The implementation of this algorithm very much similar to brute force at most k transactions only difference is that we are not limiting the number of transactions

Time Complexity:
1.Here we have n number of days and every node gives raise 2 other nodes i.e when we are in buy state we can either sell or hold and when we are in sell state we can either buy after a cooldown or cooldown which essential represents a binary tree
2. A binary tree with n number of levels(here n number of levels represents n days) have 2^n number of nodes
3. In m*n/2 number of nodes we are in sell state where we check stock prices of m stocks so the number of iterations it takes is m/2*m*n+m/2*n which can be written as Θ(m *2^n)

Space Complexity:
If we don't consider space taken by the internal stack, It takes O(n) time for storing all the possible buy and sell day values

Algorithm 8:Design a Θ(m *n2) time dynamic programming algorithm for solving Problem3

Design/Algorithm:
1.In algorithm 7 we have a time complexity of m*2^n if we see the recursion tree we are repeatedly computing values for given day and prev buy index
2. So what we can do is when get a combination of day and prev buy index we just store in dp array and based whether it's a buy cycle or sell cycle and when that index comes again in the recursion we simply return that value stored instead computing the entire thing again
3. And also while the dp table should have different index to store to buy of certain day and prev buy cause buy state and sell state at given day and prevBuy gives different output

Formula
dp[day][prevBuy][buy]=maxProfit;

Correctness of Algorithm:
Since we are just storing the buy and prevIndex that are already computed the algorithm works in all the cases

Time Complexity:
This algorithm reduces the time complexity form m*2^n to m*n^2 cause only computed a certain value when it's day prev buy is not already present
The number of combination of day and prevBuy are n*n as days range is 1-n and prevBuy can also be form 1-n and each iteration we are checking prices of m stocks so time complexity if m*n^2

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

Design/Algorithm:

Base cases,
$OPT_{sell}(i, 1) = 0$, for day 1
$OPT_{sell}(i, 1) = 0$, for stock 0
$OPT_{buy}(i, 1) = -prices[i][1]$, for day 1
$OPT_{buy}(i, 1) = 0$, for stock 0

$OPT_{sell}(i, j) = \max \{ OPT_{buy}(i, j - 1) + prices[i][j], OPT_{sell}(i, j - 1), OPT_{sell}(i - 1, j) \}$
$OPT_{buy}(i, j) = \max\{ \max_{1<=x<=m}\{OPT_{sell}(x, j-c-1)\} - prices[i][j], OPT_{buy}(i, j - 1)) \}$

Solution: $OPT_{sell}(m, n)$

1. We are given price of m stocks for n days in m*n matrix and any number of transactions is allowed.
2. For each buying stock we subtract the price and for each selling stock we add the price.
3. We consider a M[m][n] matrix which will store the optimal after sell price result (max profit) M[i][j], for i stocks and j days.
4. We consider a N[m][n] matrix which will store the optimal after buy price result N[i][j], for i stocks and j days.
5. Base case, for every i=0 to m, M[i][1] = 0, as the profit would be 0 for day 1.
6. Base case, for every j=0 to n, M[0][j] = 0, as the profit would be 0 for 0 stocks.
7. Base case, for every i=0 to m, N[i][1] = -prices[j][1], as the price would be negative if we buy at day 1.
8. Base case, for every j=0 to n, N[0][j] = 0, as the buy price would be 0 for 0 stocks.
9. For calculating M[i][j] for every i from 1 to m and j from 1 to n, we use recursion.
10. M[i][j] = max {compute-opt-buy(i, j - 1) + prices[i][j], compute-opt-sell(i, j - 1), compute-opt-sell(i - 1, j)))
11. For calculating N[i][j] for every i from 1 to m and j from 1 to n, we use recursion.
12. N[i][j] = max{ $\max_{1<=x<=m}${compute-opt-sell(x, j-c-1)} - prices[i][j], compute-opt-buy(i, j - 1)) }
13. As computing $\max_{1<=x<=m}${compute-opt-sell(x, j-c-1)}, will take another m iterations, we store the max values in X[j].
14. X[j] stores the maximum price after sell at day j considering all the stocks.
15. X[j] = $\max_{1<=x<=m}${compute-opt-sell(x, j-c-1)}
16. M[m][n] will have the resultant maximum profit.

17. To backtrack the solution, we traverse through M and N from M[m][n], to get the stock number, buy day and sell day for each transaction.

Correctness of Algorithm:

1. We will prove this by loop invariant.
2. Let M[i][j] be the maximum profit after selling a stock till i for $1 \leq i \leq m$ and day j for $1 \leq j \leq n$.
3. Let N[i][j] be the maximum amount of money agined/lost after buying stock till i for $1 \leq i \leq m$ and day j for $1 \leq j \leq n$.
4. For Initialisation, for zero stocks the maximum profit will be zero, i.e., M[0][j] = 0, for all $1 \leq j \leq n$ and, for first day, as only one stock could be bought and no stock can be sold, the maximum profit will be zero, i.e., M[i][1] = 0, for all $1 \leq i \leq m$.
5. For Initialisation, for zero stocks the maximum profit will be zero, i.e., N[0][j] = 0, for all $1 \leq j \leq n$ and, for first day, as only one stock could be bought and no stock can be sold, the price will be negative price amount, i.e., N[i][1] = -prices[i][1], for all $1 \leq i \leq m$.
6. For maintenance, consider M[i][j] for some $0 \leq i \leq m$ and $0 \leq j \leq n$. We denote M[i-1][j] as maximum profit terminating at stock i-1 and M[i][j-1] as maximum profit terminating at day j-1. If prices[i][j] is the maximum price of stock i till now, the maximum profit becomes N[i][j-1] + prices[i][j], i.e., the addition to maximum price after buying stock and the current stock price. If that is not the case, the maximum stock price would be before day j, therefore we consider M[i][j-1] or else, the maximum profit might be of a stock before i, then we consider M[i-1][j]. Hence, for every stock i and day j, the problem narrows down to finding a maximum between just three numbers: M[i-1][j], M[i][j-1] and N[i][j-1] + prices[i][j]. It therefore exhibits the optimal substructure and holds the maintenance condition.
7. For maintenance, consider N[i][j] for some $0 \leq i \leq m$ and $0 \leq j \leq n$. We denote N[i-1][j] as maximum price after buying stock terminating at stock i-1 and M[i][j-1] as price after buying stock terminating at day j-1. If prices[i][j] is the minimum price of stock i till now, the maximum price becomes $\max_{1<=x<=m}$\{M[x, j-c-1]\} - prices[i][j] (considering sell date j-c-1 before buy date), i.e., the difference of maximum price after selling stock and the current stock price. If that is not the case, the maximum stock price would be before day j, therefore we consider N[i][j-1] or else. Hence, for every stock i and day j, the problem narrows down to finding a maximum between just two numbers: N[i][j-1] and $\max_{1<=x<=m}$\{M[x, j-c-1]\} - prices[i][j]. It therefore exhibits the optimal substructure and holds the maintenance condition.
8. As the maintenance condition is followed repeatedly, the condition holds true during termination too. Since when we have $i = m + 1$, $j = n + 1$ we will already have the maximum profit for $i = m$ and $j = n$ as expected.

Time Complexity:

1. We calculate M[i][j], we do constant time operation for each recursive call. As we are using memoization, the time complexity is O(n * m).
2. We calculate N[i][j], we do constant time operation for each recursive call. As we are using memoization, the time complexity is O(n * m).
3. We calculate X[j], we iterate through 1 till m and get the maximum. As we are using memoization, the time complexity is O(n).
4. For backtracking, to traverse through the elements in M[m][n], it takes <m*n recursive calls, therefore the time complexity is O(m * n).
5. For backtracking, to traverse through the elements in N[m][n], it takes <m*n recursive calls, therefore the time complexity is O(m * n).
6. Therefore the overall time complexity comes up to O(m * n).

Space Complexity:

The algorithm uses extra space, i.e matrix to store optimal maximum profit M[m][n] and matrix to store the optimal price after buying a stock in N[m][n] and array X[n] which store the maximum sell price for each stock, hence the space complexity is O(row.column) = O(m * n).

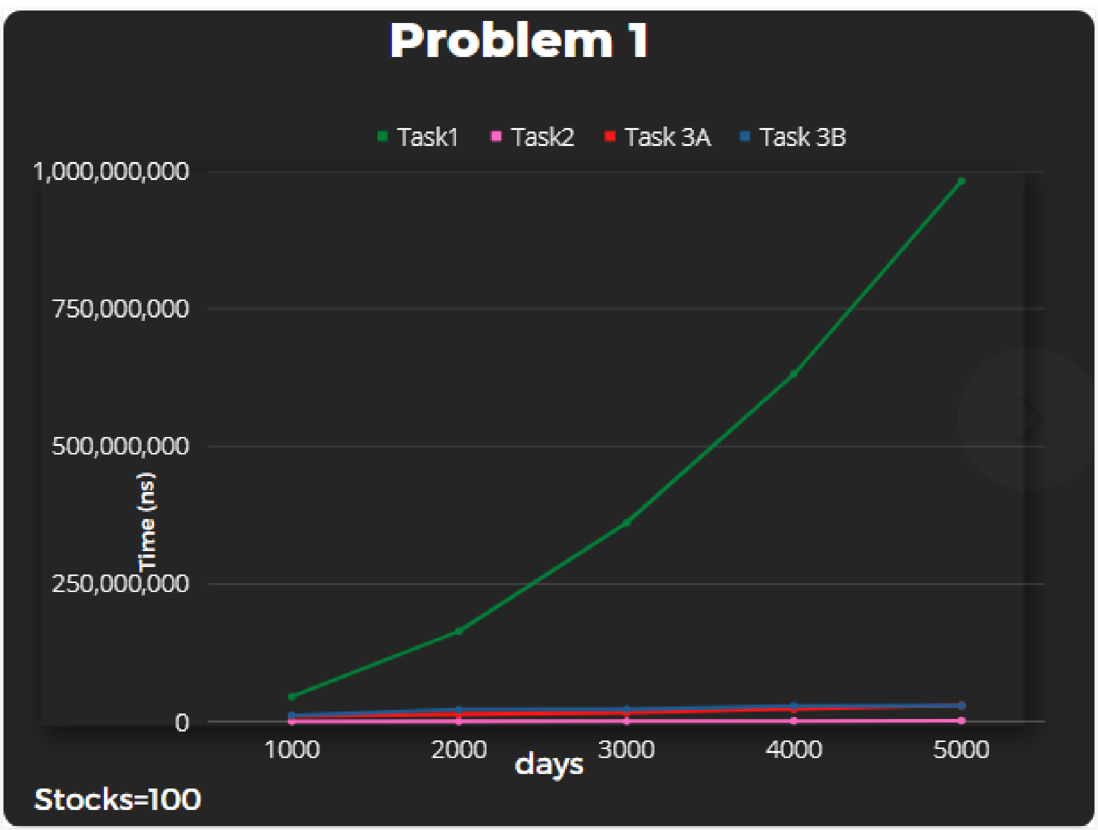
Experimental Comparative Study:

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

The below plot is for variable n that is number of days (as seen on x -axis) and fixed value of m that is the stocks, where stocks = 100. The execution time here is in nanoseconds.

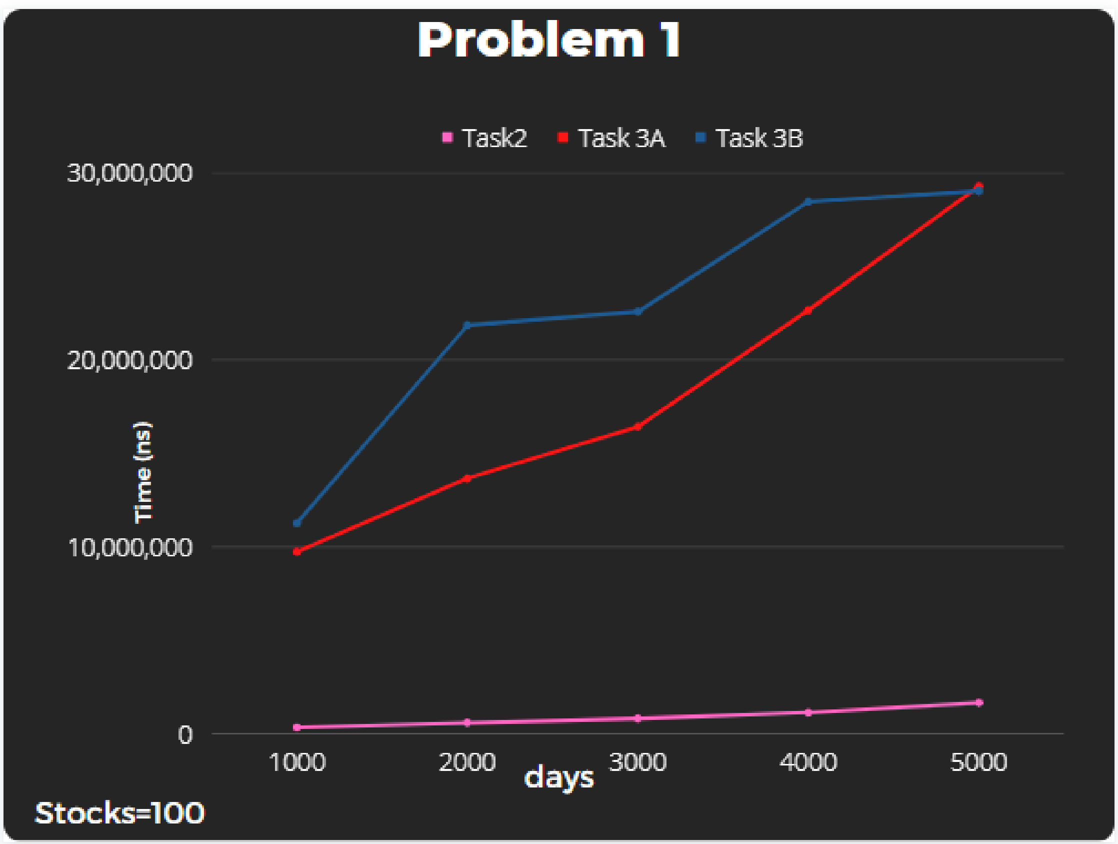
X axis - variable days
Y axis - running time.

Days Size	Task 1 (Time)	Task 2 (Time)	Task 3A(Time)	Task 3B(Time)
1000	45256900	360300	9744500	11265900
2000	164144200	607200	13667700	21843500
3000	361119600	835300	16422800	22569700
4000	631429000	1154800	22648500	28450600
5000	981308700	1674500	29264300	29012700



Below is a plot for Task2 , Task3A, Task 3B with varied days and fixed stocks.

Days Size	Task 2 (Time)	Task 3A(Time)	Task 3B(Time)
1000	360300	9744500	11265900
2000	607200	13667700	21843500
3000	835300	16422800	22569700
4000	1154800	22648500	28450600
5000	1674500	29264300	29012700

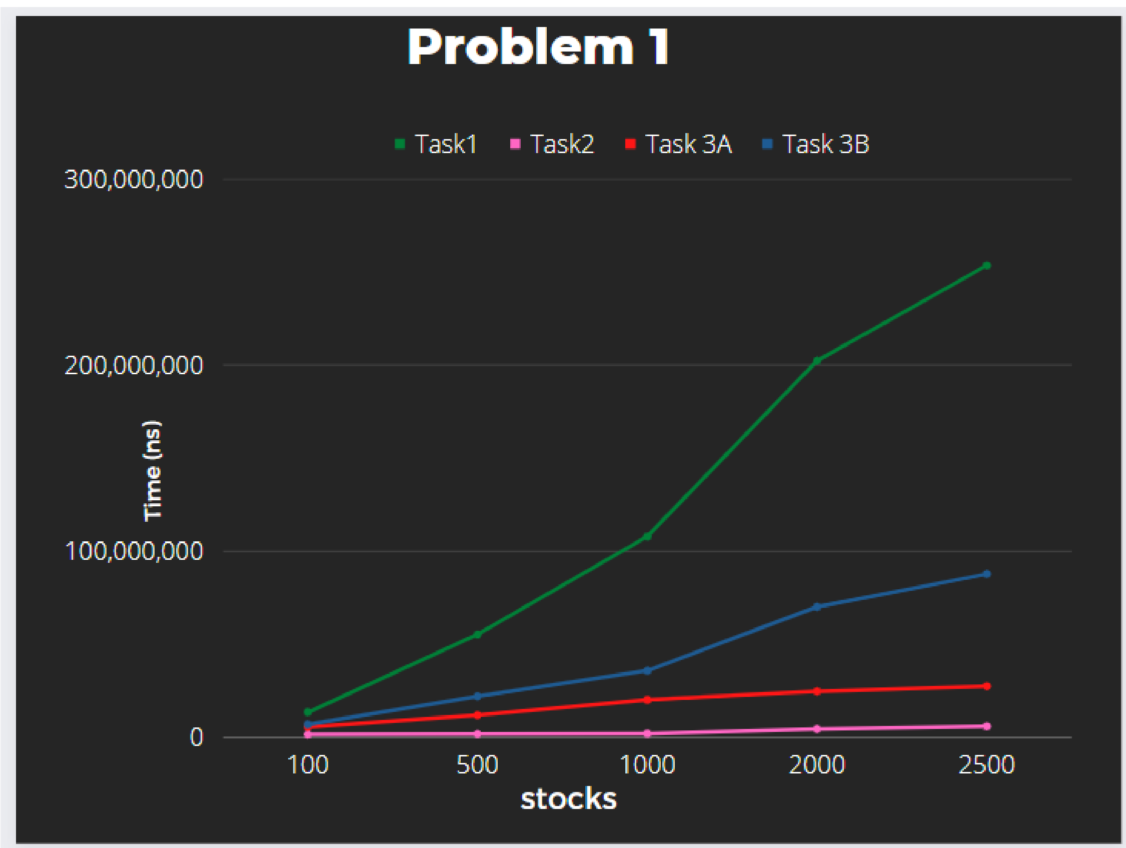


Plot2 Comparison of Task1, Task2, Task3A, Task3B with variable m and fixed n.

The below plot is for variable m that stocks (as seen on x -axis) and fixed value of n that is the days, where days= 100. The execution time here is in nanoseconds.

X axis - variable stock.
Y axis - running time.

Stock Size	Task 1 (Time)	Task 2 (Time)	Task 3A(Time)	Task 3B(Time)
10	13662700	1632200	5702900	7062500
500	55377100	1897300	12003700	22188800
1000	107944000	2155500	20337500	35875300
2000	202270800	4587300	24857900	70180600
2500	253536400	6073200	27574400	87824000



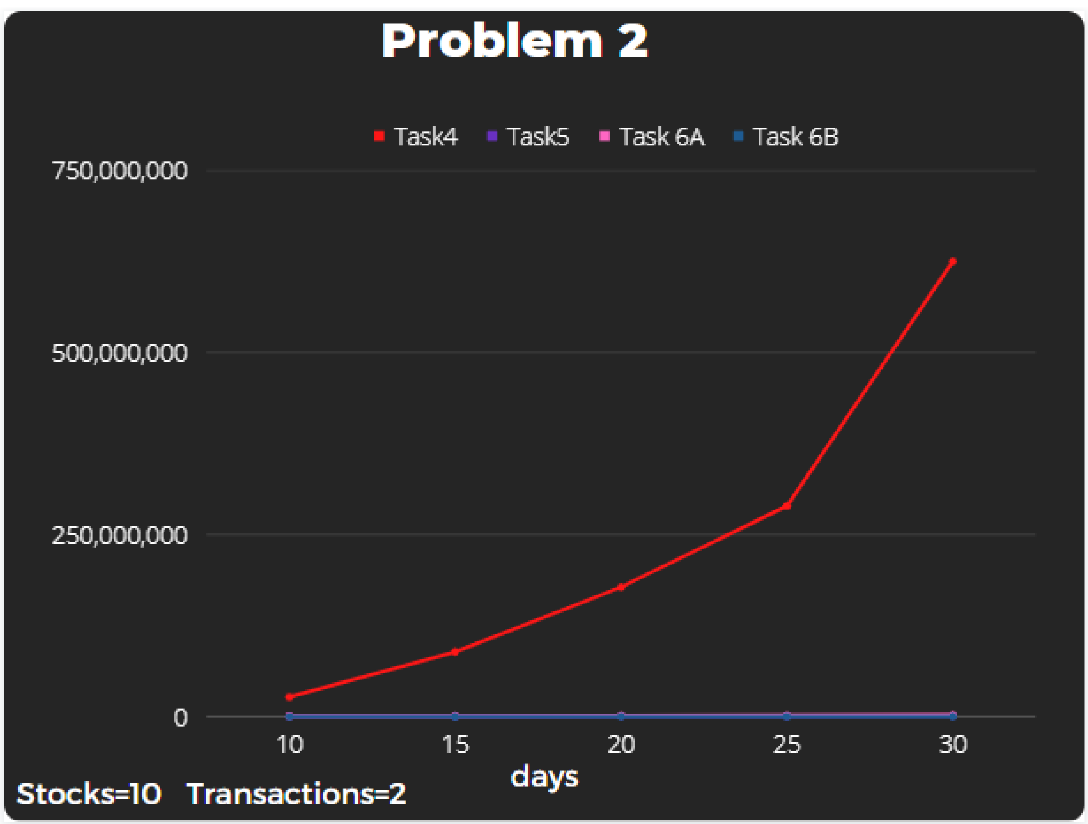
Plot3

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

The below plot is for variable n that days(as seen on x -axis) and fixed value of m that is the stocks, where m = 100 and fixed value of k = 2 transactions. The execution time here is in nanoseconds.

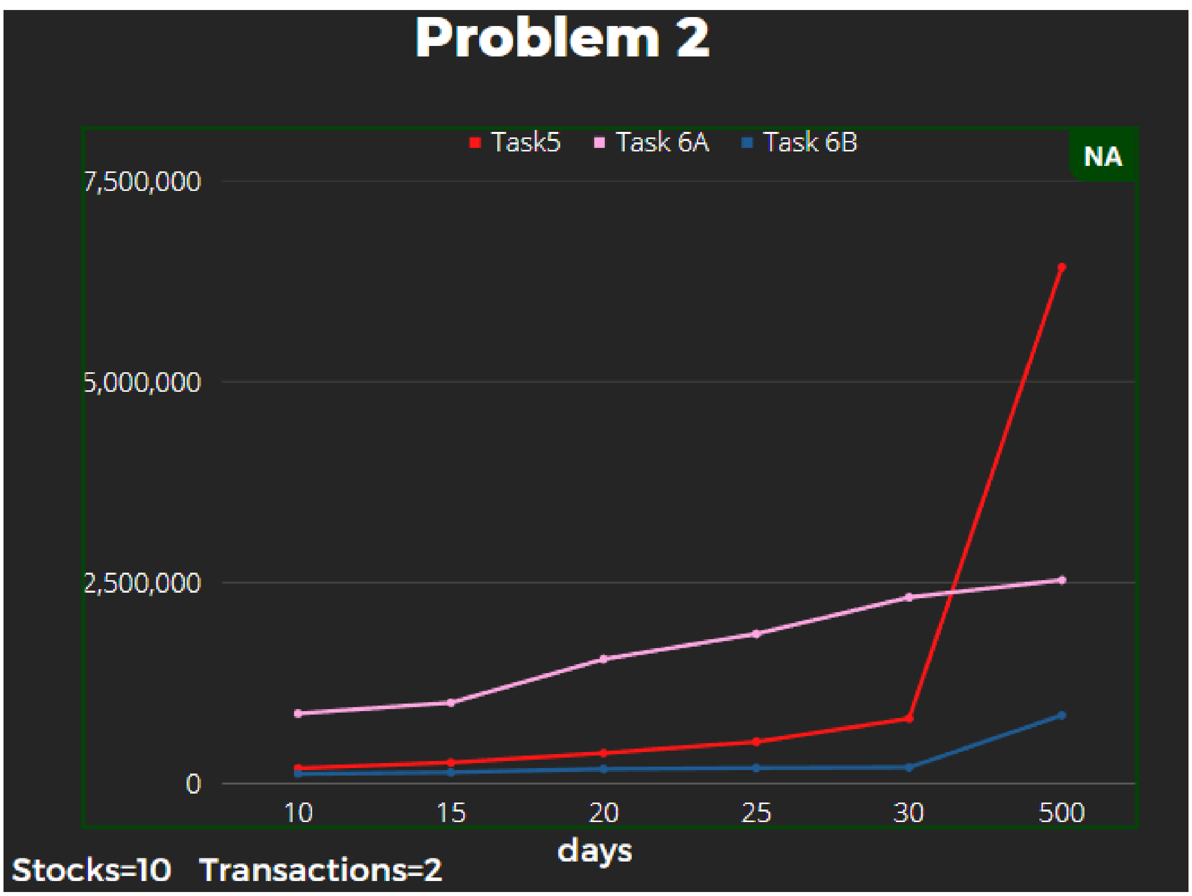
X axis - variable days.
Y axis - running time.

Days Size	Task 4 (Time)	Task 5 (Time)	Task 6A(Time)	Task 6B(Time)
10	26924900	191800	873400	123000
15	89142500	264300	1007700	141500
20	177897100	380700	1548200	184100
25	289318600	520900	1863500	196500
30	624839200	810700	2317200	203400



Below is a plot for Task5 , Task 6A , Task 6B for varied days, fixed transactions and fixed stocks.

Input Size	**Task 5 (Time)**	**Task 6A(Time)**	**Task 6B(Time)**
10	191800	873400	123000
15	264300	1007700	141500
20	380700	1548200	184100
25	520900	1863500	196500
30	810700	2317200	203400
500	6422500	2532400	851600

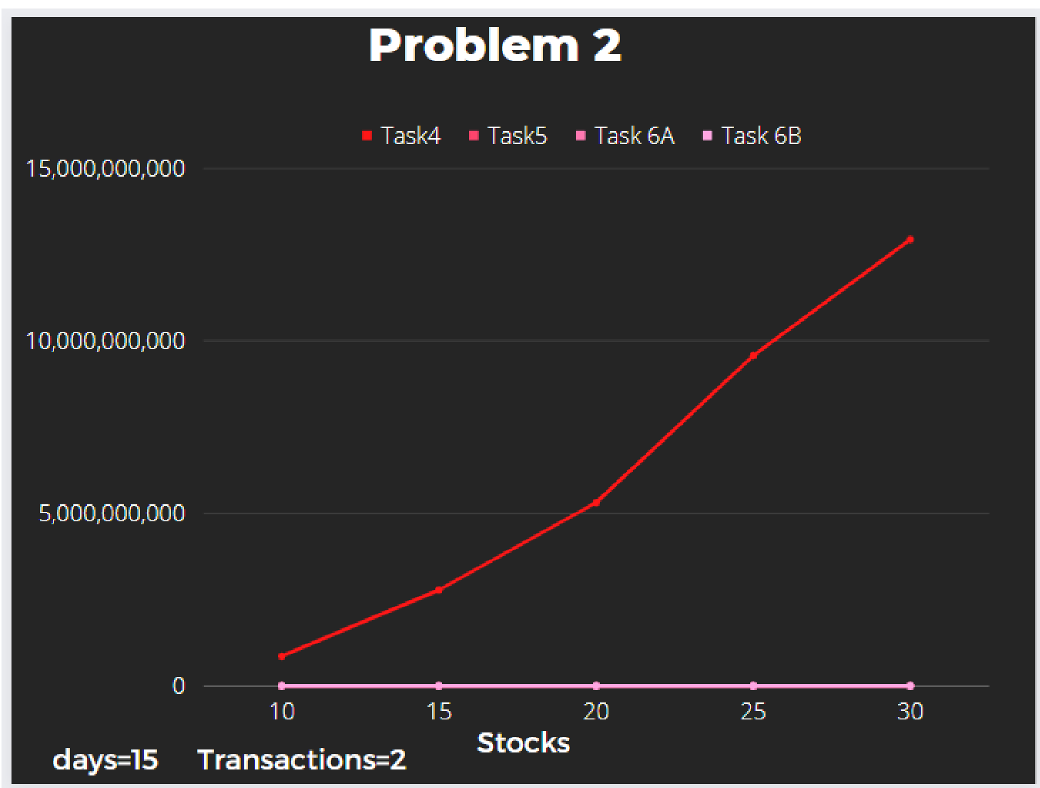


Plot4
Comparison of Task4, Task5, Task6A, Task6B with variable m and fixed n and k.

The below plot is for variable m that stocks (as seen on x -axis) and fixed value of n that is the days , where n = 15 and fixed value of k = 2 transactions. The execution time here is in nanoseconds.

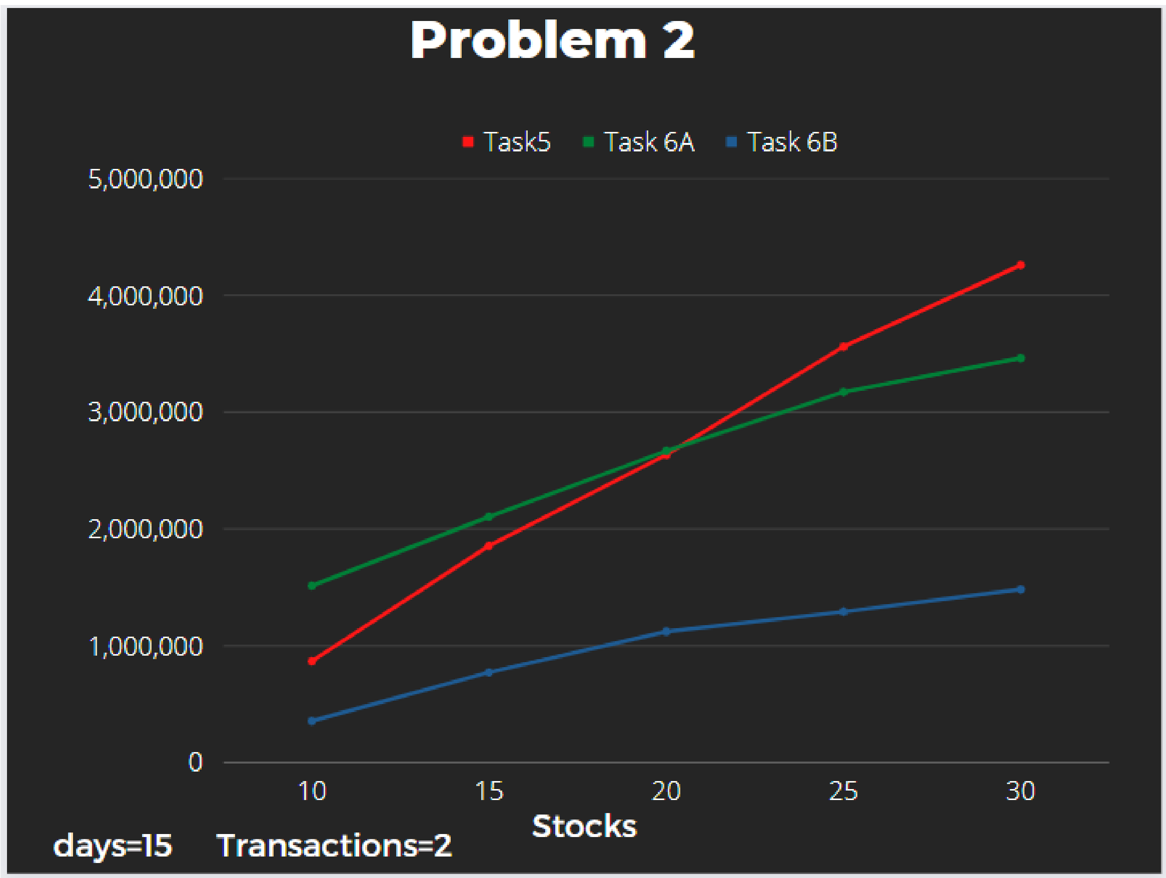
X axis - variable stocks.
Y axis - running time.

Days Size	Task 4 (Time)	Task 5 (Time)	Task 6A(Time)	Task 6B(Time)
10	860224200	867300	1513500	355500
15	2777084100	1855300	2104200	772100
20	5306727200	2633900	2667300	1121400
25	9563748600	3559100	3171600	1291800
30	12935789900	4257500	3462700	1483700



Below is a plot of Task 5, Task 6A and Task 6B, with varied stocks and fixed number of transactions and days.

Input Size	**Task 5 (Time)**	**Task 6A(Time)**	**Task 6B(Time)**
10	867300	1513500	355500
15	1855300	2104200	772100
20	2633900	2667300	1121400
25	3559100	3171600	1291800
30	4257500	3462700	1483700

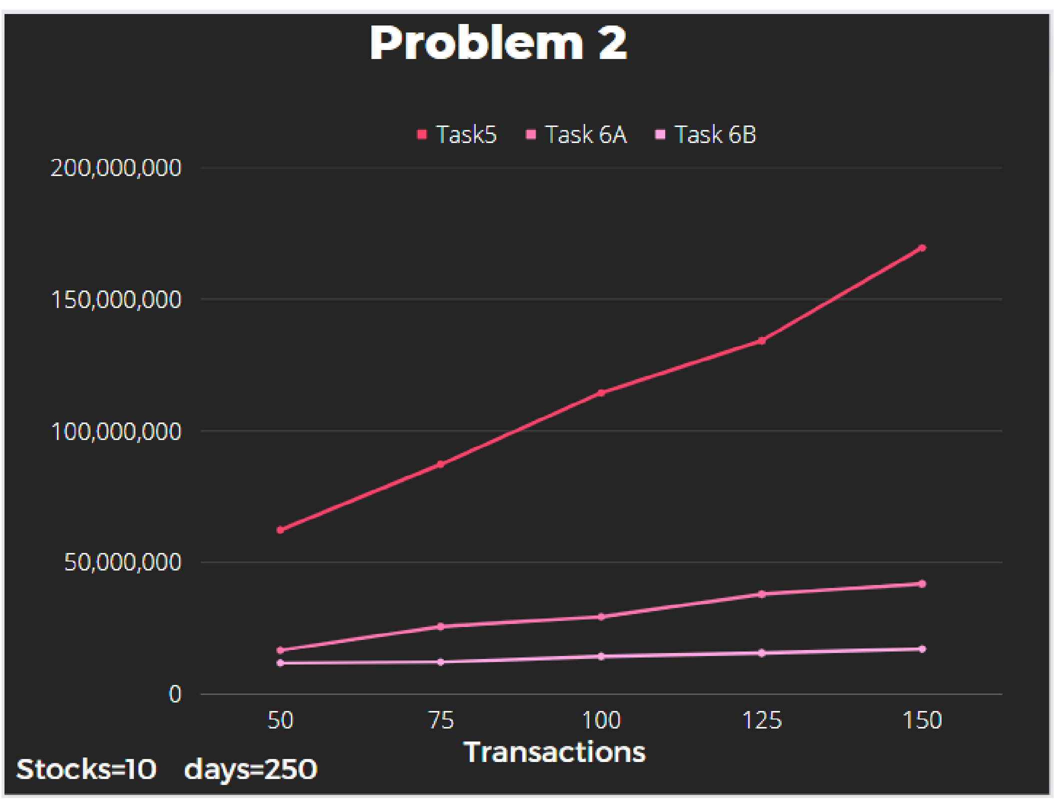


Plot5
Comparison of Task4, Task5, Task6A, Task6B with variable k and fixed m and n

The below plot is for variable k that is the number of transactions (as seen on x -axis) and fixed value of n that is the days ,where n = 15 and fixed value of m = 2 stocks. The execution time here is in nanoseconds.

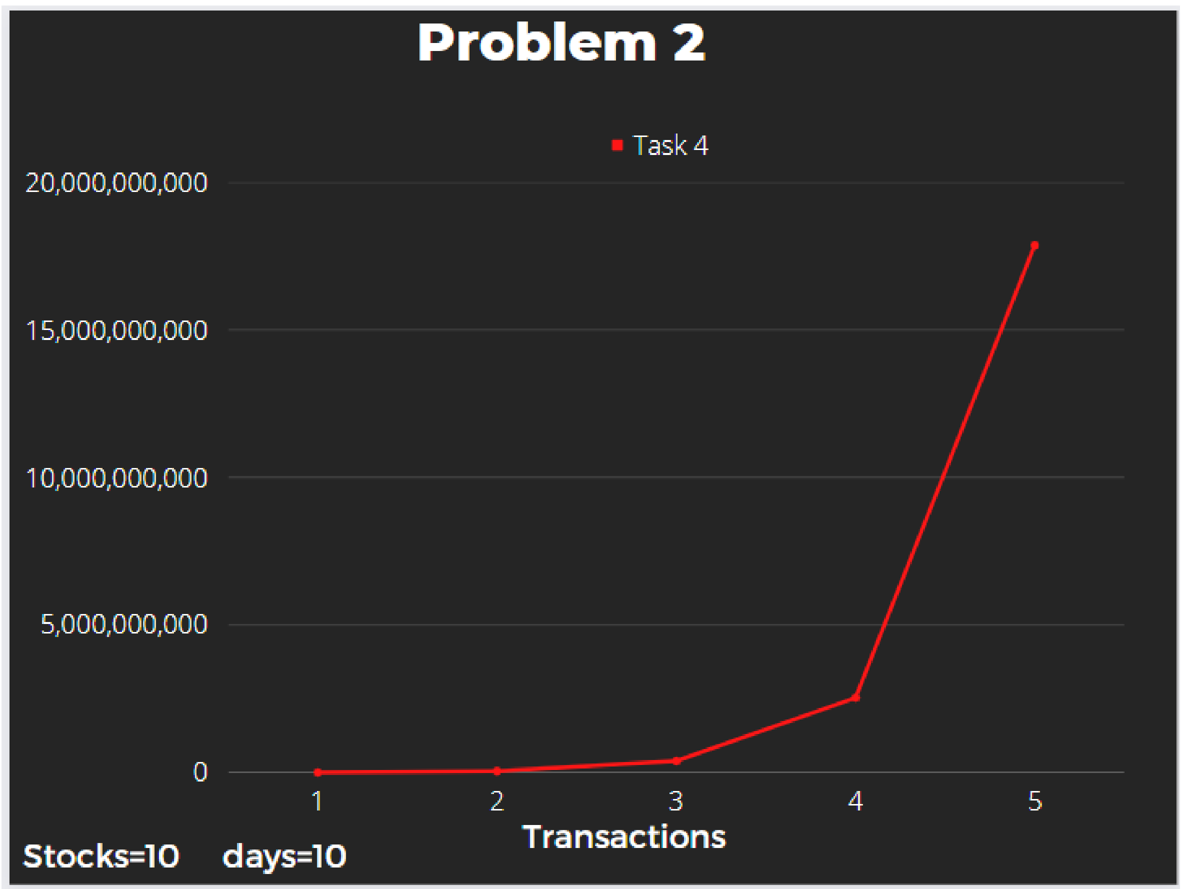
X axis - variable transactions.
Y axis - running time.

K Size	Task 5 (Time)	Task 6A(Time)	Task 6B(Time)
50	62281000	16592100	11774900
75	87250500	25610500	12180300
100	114372800	29293200	14316900
125	134160900	37812900	15522600
150	169469600	41850300	17119300



Below is a plot of Task 4, with varied transactions and fixed number of stocks and days.

K Size	Task 4 (Time)
1	2528000
2	41295100
3	386876600
4	2525534800
5	17867018000



Output Screenshots:

Algorithm 1 - Task 1:

Administrator: Command Prompt

```
C:\Users\Priti Gumaste\Desktop\AOA1>make
javac -g Task1.java
java Task1
Enter number of stocks:
3
Enter number of days:
5
Enter the matrix values:
1 7 4 0 9

4 8 8 2 4

5 5 1 7 1
Profit is: 6

C:\Users\Priti Gumaste\Desktop\AOA1>_
```

Algorithm 2 - Task 4:

Administrator: Command Prompt

```
C:\Users\Priti Gumaste\Desktop\AOA1>make
javac -g Task4.java
Note: Task4.java uses unchecked or unsafe operations.
Note: Recompile with -Xlint:unchecked for details.
java Task4
Enter number of stocks:
6
Enter number of days:
9
Enter number of transactions:
3
Enter elements of the matrix:
91 27 68 22 45 24 59 97 71

16 28 40 75 66 41 90 47 79

40 77 49 25 22 47 64 46 59

64 27 30 63 17 22 64 35 7

16 61 71 12 49 85 82 7 23

38 54 22 54 93 27 33 49 44
205
1 0 3
4 3 5
0 5 7
```

Algorithm 2 - Task 5:

```
C:\Users\Priti Gumaste\Desktop\AOA1>make
javac -g Task5.java
java Task5
Enter Number of Stocks
3
Enter Number of Days
5
Enter Number of Transactions
3
Enter Stock Prices for M stocks in N days
1 7 4 0 9

4 8 8 2 4

5 5 1 7 1
0 0 0 0 0
0 0 0 0 0
0 0 0 0 0
0 0 0 0 0

0 0 0 0 0
0 6 6 6 9
0 6 6 6 9
0 6 6 6 9

0 0 0 0 0
0 6 6 6 15
0 6 6 6 15
0 6 6 12 15

0 0 0 0 0
0 6 6 6 21
0 6 6 6 21
0 6 6 12 21

Buy Stock 1 at day 0
Sell Stock 1 at day 1
Buy Stock 3 at day 2
Sell Stock 3 at day 3
Buy Stock 1 at day 3
Sell Stock 1 at day 4

C:\Users\Priti Gumaste\Desktop\AOA1>
```

Algorithm 2 - Task 6B:

```
C:\Users\Priti Gumaste\Desktop\AOA1>make
javac -g Task6B.java
java Task6B
Enter Number of Stocks
3
Enter Number of Days
10
Enter Number of Transactions
4
Enter Stock Prices for M stocks in N days
4 13 94 22 41 21 65 66 1 6

68 8 79 8 45 13 79 71 22 16

16 6 67 78 25 15 6 73 15 50
0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0
0 9 90 90 90 90 90 90 90 90
0 9 90 90 90 90 90 90 90 90
0 9 90 90 90 90 90 90 90 90
0 0 0 0 0 0 0 0 0 0
0 9 90 90 109 109 134 135 135 135
0 9 90 90 127 127 161 161 161 161
0 9 90 101 127 127 161 161 161 161
0 0 0 0 0 0 0 0 0 0
0 9 90 90 120 120 171 172 172 172
0 9 90 90 138 138 193 193 193 193
0 9 90 101 138 138 193 228 228 228
0 0 0 0 0 0 0 0 0 0
0 9 90 90 120 120 182 194 194 233
0 9 90 90 138 138 204 204 204 233
0 9 90 101 138 138 204 260 260 263
Buy Stock 1 at day 0
Sell Stock 1 at day 2
Buy Stock 2 at day 3
Sell Stock 2 at day 6
Buy Stock 3 at day 6
Sell Stock 3 at day 7
Buy Stock 3 at day 8
Sell Stock 3 at day 9

C:\Users\Priti Gumaste\Desktop\AOA1>
```

Conclusion:

1. After considering the performance metrics for all the different complexities of various implementations of the algorithms, we learn that brute force is never an optimal way of implementation.
2. Through the experimentative study we infer that as we increase the input size for these algorithms the time taken to execute them also increases drastically. It can be concluded that the dynamic way of implementation is an optimal way as the problem is divided into multiple small problems and solved using various approaches such as bottom up and recursive implementation.
3. Task1: For problem 1, this was the brute force implementation that took us O(m*n^2) time complexity. It was iterative with nested loops that took us the extra time. Brute force is a very easy approach to come up with which is not the optimal approach as well, hence the increase in time complexity. As we tried with large input, the curve went above and took a lot of time to get the output.
4. Task2: This was the greedy approach that gave us similar time complexity as the dynamic approach as we just had to output the maximumProfit. The time complexity was O(m*n).
5. Task3: For the dynamic programming implementation, it took us a while to figure out the bottom up and the memoization part. This had a better time complexity as that of greedy implementation, i.e, O(m*n).
6. Task4: Brute force for the second question was the most difficult one to figure out, the time complexity was the worst with O(m*n^2*k).
7. Task5: The dynamic approach was also a challenging task for us, as this involved a multi-dimensional approach and we had to keep track of the indices of the transactions that gave us the maximum profit. This had the time complexity of O(m*n^2*k).
8. Task6: The dynamic implementation seemed like a very challenging experience for us as we had to output the k transactions that gave us the maximum profit, that involved using memoization and the bottom-up approach.
9. Task7: Brute force approach of the third problem had the time complexity of O(m*2^n).
10. Task8: The dynamic implementation of the problem was a difficult one to implement with the time complexity of O(m*n^2).
11. Task9: The recursive and the iterative implementation had better time complexity than the other two implementations of the third problem. Using memoization and bottom up approach we were able to implement it in O(m*n) time complexity.

References:

1. https://leetcode.com/problems/best-time-to-buy-and-sell-stock-with-cooldown/
2. https://www.geeksforgeeks.org/maximum-profit-by-buying-and-selling-a-share-at-most-k-times/
3. https://www.geeksforgeeks.org/stock-buy-sell/
4. https://www.youtube.com/watch?v=oDhu5uGq_ic