PROJECT DESCRIPTION

1 Problem Definition

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 $i = 1, \ldots, m$ and $j = 1, \ldots, n$. 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 $m \times n$ integers (non-negative) representing the predicted prices of m stocks for n days, find a single transaction (buy and sell) that gives maximum profit.

PROBLEM2 Given a matrix A of $m \times n$ 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.]

PROBLEM3 Given a matrix A of $m \times n$ 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 number of transactions. However, you cannot buy any stock for c days after selling any stock. If you sell a stock at day i, you are not allowed to buy any stock until day $i + c + 1$

2 Algorithm Design Tasks

You are asked to design three different algorithms for each problem, with varying time c plexity requirement in order to conduct an experimental comparative study.

ALG1 Design a $\Theta(m * n^2)$ time brute force algorithm for solving PROBLEM1

ALG2 Design a $\Theta(m * n)$ time greedy algorithm for solving PROBLEM1

ALG3 Design a $\Theta(m * n)$ time dynamic programming algorithm for solving PROBLEM1

ALG4 Design a $\Theta(m * n^{2k})$ time brute force algorithm for solving PROBLEM2

ALG5 Design a $\Theta(m * n^2 * k)$ time dynamic programming algorithm for solving PROBLEM2

ALG6 Design a $\Theta(m * n * k)$ time dynamic programming algorithm for solving PROBLEM2

ALG7 Design a $\Theta(m * 2^n)$ time brute force algorithm for solving PROBLEM3

ALG8 Design a $\Theta(m * n^2)$ time dynamic programming algorithm for solving PROBLEM3

ALG9 Design a $\Theta(m * n)$ time dynamic programming algorithm for solving PROBLEM3

3 Programming Tasks

Once you complete the algorithm design tasks, perform the following programming tasks:

TASK1 Give an implementation of ALG1.

TASK2 Give an implementation of ALG2.

TASK3A Give a recursive implementation of ALG3 using **Memoization**.

TASK3B Give an iterative **BottomUp** implementation of ALG3.

TASK4 Give an implementation of ALG4.

TASK5 Give an implementation of ALG5.

TASK6A Give a recursive implementation of ALG6 using **Memoization**.

TASK6B Give an iterative **BottomUp** implementation of ALG6.

TASK7 Give an implementation of ALG7.

TASK8 Give an implementation of ALG8.

TASK9A Give a recursive implementation of ALG9 using **Memoization**.

TASK9B Give an iterative **BottomUp** implementation of ALG9.

4 Language/Input/Output Specifications

You may use C++, Java. Your program must compile/run on the Thunder CISE server using gcc/g++ or standard JDK. You may access the server using SSH client on `thunder.cise.ufl.edu`. You must write a `makefile` document that creates an executable named **Stocks**. The task is passed by an argument, e.g., when **Stocks 3b** is called from the terminal, your program needs to execute the implementation of TASK3B. Through out this assignment assume that the indices of the stocks are $1 \dots m$ and the indices of the days are $1 \dots n$.

PROBLEM1: For convenience assume that $1 \leq m < 100$, $1 \leq n < 10^5$ and $\forall i \;\; 0 \leq A[i][j] < 10^4$. If multiple buy/sell transaction pairs yield the maximum profit, output any one of them.

Input. Your program will read input from standard input (`stdin`) in the following order:

- Line 1 consists of two integers m and n separated by a single space.
- Next m lines each consists of n integers (prices for n days) separated by a single space.

Output. Print the optimal transaction info to standard output (`stdout`)

- A single line with 3 integers (*Stock, BuyDay,* & *SellDay* indices) separated by a space.

PROBLEM2: For convenience assume that $1 \leq k < 100$, $1 \leq m < 100$, $1 \leq n < 1000$ and $\forall ij \;\; 0 \leq A[i][j] < 1000$. If multiple sets of transactions yield the maximum profit, output any one of them.

Input. Your program will read input from standard input (`stdin`) in the following order

- Line 1 consists one integer k.
- Line 2 consists of two integers m and n separated by one space character.
- Next m lines each with n integers (predicted prices) separated by a single space.

PROBLEM 3

For convenience assume that $1 \leq c < 100$, $1 \leq m < 100$, $1 \leq n < 1000$ and $\forall i \;\; 0 \leq A[i][j] < 1000$. If there are multiple possible sets of transactions that yield the maximum profit, output any one of them.

Input. Your program will read input from standard input (`stdin`) in the following order:

- Line 1 consists one integer c.
- Line 2 consists of two integers m and n separated by one space character.
- Next m lines each with n integers (predicted prices) separated by a single space.

Output. Print a set of transactions that yields the max profit in order of dates.

- Each line with 3 integers (*Stock, BuyDay,* & *SellDay* indices) separated by a single space.

STEPS TO RUN CODE

1. Navigate to the directory which contains the zip file.
2. Unzip the zip file using: “unzip filename.zip”
3. Run the commands to

   A. Command to take input from user

   > make

   > java Stocks taskname

   Example: java Stocks 3a

   OR

   B. Command to take input from a text file

   > make

   > java Stocks taskname filename

   Example: java Stocks 3a input.txt

DESIGN AND ANALYSIS OF ALGORITHMS

ALGORITHM 1:

a. DESIGN

Definitions:

m = Number of stocks

n = Number of days

stocks =m xn matrix containing m stock for n days

Design(Considering 1 based index):

```
finalMaxProfit = -INF
finalBuyIndex = -1
finalSellIndex = -1
stock = -1
for i = 1 to m
        maxProfit = -INF
        for j = 1 to n-1
                for k = j+1 to  n
                        profit = stocks[i][k] - stocks[i][j]
                        if profit > maxProfit
                                maxProfit = profit
                                buyIndex = j
                                sellIndex =k
         if maxProfit >  finalMaxProfit
                finalMaxProfit = maxProfit
                stock = i
                finalBuyIndex = buyIndex
                finalSellIndex = sellIndex

return stock ,finalBuyIndex, finalSellIndex
```

b. CORRECTNESS

We will be proving the correctness of this algorithm by using the below loop invariant.

Loop Invariant:

When we are at the start of iteration i+1(1 to m) and iteration j+1 (1 to n-1), the variable finalMaxProfit contains the maximum profit of the 2D matrix[1…i][1…j] with the stock value, buyDay and sellDay.

Initialization:

During the very first iteration, i will be equal to 1 and j will be equal to 1, so the maximum profit in finalMaxProfit will be stocks[1][2] – stocks[1][1] as the stored maxProfit is currently having Integer.Min_value. Thus, the invariant is satisfied for the next iteration.

Maintenance:

Let's assume that the loop invariant holds true at the start of iteration of p+1 in loop (i=1…m) and iteration of q+1 in loop (j=1…n-1)

So, according to our loop invariant, finalMaxProfit will contain the maximum profit of our 2D matrix stocks[1..p][1..q]. There will be two cases here

1. When the stocks[p][q] > finalMaxProfit, in that case the maxProfit will be updated and now, stocks[p][q] will be maximum for stocks[1…p][1..q]. Thus, it will satisfy the loop invariant for the next loop as the maxProfit will have the maximum at the start of the next loop.
2. When the stocks[p][q] < maxProfit, in that case the maxProfit will stay the same Thus, it will satisfy the loop invariant for the next loop as the maxProfit will have the maximum at the start of the next loop.

Termination:

The algorithm will terminate once the iteration i=m and j =n-1 is completed, thats when we will have the maximum for stocks[1..m][1…n].

c. TIME COMPLEXITY

Worst Case Time :O(m*n^2)

Where

m = number of stocks

n = number of days

We iterate over m stocks which gives a time complexity of O(m) and for each stock we find the maximum profit by considering every day as buy day and all next indexes as sell day which which gives a time complexity of O(n^2)

So as we can see 3 nested loops we have a time complexity of 0(m*n^2)

d. SPACE COMPLEXITY

Space Complexity :O(1)

We do not use any storage space with respect to n hence we have an space complexity of O(1)

ALGORITHM 2:

a.DESIGN

Definitions:

m = Number of stocks

n = Number of days

stocks =m x n matrix containing m stocks for n days

Design:

```
finalMaxProfit = -INF
finalBuyIndex = -1
finalSellIndex = -1
stock = -1
for i = 1 to m
        minPrice= +INF
        maxProfit = -INF
        for j = 1 to n
                if stocks[i][j] < minPrice
                        minPrice = stocks[i][j]
                        buyIndex = j
                else if stocks[i][j] - minPrice> maxProfit
                        maxProfit = stocks[i][j] - minPrice
                        sellIndex = j
         if maxProfit >  finalMaxProfit
                finalMaxProfit = maxProfit
                stock = i
                finalBuyIndex = buyIndex
                finalSellIndex = sellIndex

return stock ,finalBuyIndex, finalSellIndex
```

B. CORRECTNESS

Loop Invariant:

When we are at the start of iteration i+1(1 to m) and iteration j+1 (1 to n-1), the variable minPrice contains the minPrice and variable maxProfit of stocks[1…i][1…j] and with the stock value, buyDay and sellDay.

Initialization:

At the start of the very first iteration, the iterators I and j are 1. So the minPrice is initialized to the largest possible integer and maxPrice is initialized to the lowest possible integer, which holds true for matrices of size 0.

Maintenance:

Let's assume that the loop invariant holds true at the start of iteration of p+1 in loop (i=1…m) and iteration of q+1 in loop (j=1…n-1)

So, according to our loop invariant, minPrice will contain the minimum price of our 2D matrix stocks[1..p][1..q] and maxProfit will contain the maximum profit for 2D matrix stocks[1...p][1…q]. Now, at the start of next iteration, there will be following cases

1. When the stocks[p][q+1] < minPrice, in that case the minPrice will be updated and now, stocks[p][q+1] will be minimum for stocks[1…p][1..q+1]. Thus, it will satisfy the loop invariant for the next loop as the minPrice will have the minimum at the start of the next loop.
2. When the stocks[p][q+1] > minPrice and stocks[p][q+1] – minPrice > maxProfit, in that case the stocks[p][q+1] – minPrice will be themaxProfit.Thus, it will satisfy the loop invariant for next loop as the maxProfit will have the maximum at the start of next loop.

Termination:

The algorithm will terminate once the iteration i=m and j =n is completed, that's when we will have the maximum for stocks[1..m][1…n].

c.TIME COMPLEXITY

Worst Case Time :O(m*n)

Where

m = number of stocks

n = number of days

We iterate over m stocks which gives a time complexity of O(m) and for each stock

We call theCOMPUTE() function recursively over all days. But we also cache the overlapping suproblems in an array and return the values which are already computed. So we make only n recursive calls in total. With a time complexity of O(n). Hence the total time complexity is O(m*n)

d.SPACE COMPLEXITY

Space Complexity :O(1)

We do not use any storage space with respect to n hence we have an space complexity of O(1)

ALGORITHM 3A:

a. DESIGN

Definitions:

m = Number of stocks

n = Number of days

stocks =m xn matrix containing m stock for n days

OPT(i):Gives the maximum profit transaction for a given stock uptill day i

OPT(n): Gives the maximum profit transaction for a given stock

OPT(m,n) :Gives maximum profit of m stocks for n days

P_{max} = max profit until now for a given stock

$P_{maxFinal}$ = max profit for all stocks

Goal:

For 1 stock:

OPT(n) = Pmax i = n …..base case

= max{Pmax , OPT(i-1)} otherwise

1<=i<=n

As OPT(i-1) gives the maxProfit for i-1 days so we find max till now and OPTi-1) to get OPT(i)

For m stocks

OPT(m,n) = max{ $P_{maxFinal}$,OPT(n) }

1<=n<=m

We can store the recursion results in an dp array

Pseudo Code:

```
finalMaxProfit = -INF
finalBuyIndex = -1
finalSellIndex = -1
stock = -1
for i = 1 to m
        Array dp[1…n+1]
        Fill dp with + INF
         COMPUTE(stockPrices[i],0,dp)
         if maxProfit >  finalMaxProfit
                finalMaxProfit =maxProfit
                stock = i
                finalBuyIndex = buyIndex
                finalSellIndex = sellIndex
```

```
return stock ,finalBuyIndex, finalSellIndex

COMPUTE(int[] prices,int[] dp, int index):
        if index == length of prices
                return

        if dp[index] > prices[index]
                dp[index+] = prices[index]
                buyDayIndex = index
        else
                dp[index+1] = dp[index]

         COMPUTE(prices,index+1,dp,output)
        if(maxProfit < prices[index] - dp[index+1])
                sellDayIndex = index
                maxProfit = prices[index] - dp[index+1]
        return maxProfit,buyIndex,sellIndex
```

b.CORRECTNESS

Proof by induction:

We consider OPT(i) will give us the maximum transaction profit for a given stock upto index i. We assume OPT(i-1) gives us the maximum profit for i-1 days. Now we can find OPT(i) by taking the maximum of all previous results. We can say that OPT(i) will be correct if OPT(i-1) gives us the correct answer.

We iterate over all the stocks and find OPT(m,n). This gives us the maximum transactions for all the available stocks present.

We consider that OPT(i-1) will; give us the best buying price from 0 to i-1 days

Termination:

The code will terminate at base case when i == stockPrices[0].length and we will start returning and the methods will start popping from the stack and we start moving up the recursive tree

c.TIME COMPLEXITY

Worst Case Time for :O(m*n)

Where

m = number of stocks

n = number of days

We iterate over stocks of size m which gives time complexity of O(m) . We make n recursive calls and there are non overlapping subproblems. Total time complexity is O(m*n)

d.SPACE COMPLEXITY

Space Complexity :O(m*n)
Recursive Stack Space Complexity :O(m*n)

We use a dp array of size n and use m such arrays so space complexity is O(m*n). Also for each of m stock we are using a recursion depth of n hence Recursive Stack Space Complexity will be O(m*n)

ALGORITHM 3B:

a.DESIGN

Defination:
m = Number of stocks
n = Number of days
stocks =m xn matrix containing m stock for n days

Pseudo Code:

```
finalMaxProfit = -INF;finalBuyIndex = -1;finalSellIndex = -1;stock = -1
for i = 1 to m
        Array dp[1..n+1]
        maxprofit = -INF
        if(stockPrices[i].length > 0)
                dp[0] = stockPrices[i]
        else
                dp[0] = -1
        for j = 0 to n
                if stockPrices[i][j] < dp[j]
                        dp[j+1] = stockPrices[i][j]
                        buyDay = j
                else
                        dp[j+1] = dp[j]
                if stockPrices[i][j] - dp[j] > maxProfit
                        maxProfit = stockPrices[i][j] - dp[j]
                        sellDay = j
         if maxProfit >  finalMaxProfit
                finalMaxProfit = maxProfit
                stock = i
                finalBuyIndex = buyIndex
                finalSellIndex = sellIndex
return stock ,finalBuyIndex, finalSellIndex
```

b.CORRECTNESS

Proof by Loop Invariant:

We keep a table to store the max profit of stock until now.

We iterate through all the stocks and find the maximum of all transactions. The maximum of all transactions will give us the maximum transaction of all the socks. Since Kadanes algorithm is an iterative algorithm we use loop invariant algorithm

Initialization

We initialise each iteration of i and j with 1.
We initialize finalMaxProfit , finalBuyIndex,finalSellIndexbefore with -1 befor the loop begins.We initialise dp array in each iteration of stock with current first day of stock or -1. So these values are correct before the loop begins

Loop Invariant

The loop iterates from i = 1 to m and j = 1 to n. maxProfit holds the maxProfit of the current stocks and finalMaxProfit holds the maximum profit of all the stocks.dp holds the maximum profit until now.We will have maximum profit transaction for all the stocks when the outer loop ends.

Mantainance:

In the indder loop we check the current price with the price in dp array. If stockPrices[i][j] < dp[j] then we buy on that day and assign dp[j+1] to stockPrices[i][j] else we reassign dp[j+1] to dp[j].If stockPrices[i][j] - dp[j] > maxProfit then we reassign maxProfit and buy on that day. So we will have the maximum transaction for each stock in the inner loop. In the outer loop we find the maximum of all the inner loop by checking with finalMaxProfit . So if this iteration holds correct values then all iterations will hold correct values. Hence the correct values are maintained in each loop iteration

Termination:

The outer loop terminates when i reaches m . At the end of the loop we have the maximum profitable transaction of all the stocks and will be present at the last index of the dp array.So the values are correct when loop ends.

c.TIME COMPLEXITY

Worst Case Time for :O(m*n)
Where
m = number of stocks
n = number of days
We iterate over stocks of size m which gives time complexity of O(m) and iterate over all days which is O(n), Total time complexity if O(m*n)

d.SPACE COMPLEXITY

Space Complexity :O(m*n)
We use a dp array of size n and use m such arrays so space complexity is O(m*n).

ALGORITHM 4

a.DESIGN

Definition:

OPT(day,k,haveStock,stockOptions,currentProfit)

k = number of transactions

haveStock = check whether you have a stock or not

stockOptions = options if taking any stockk for a given day or not taking a stock at all

currentProfit = profit until now

Algorithm:

Base Case:

```
if k = 0 OR day = stockPrices[0].length
        if day == stockPrices[0] and haveStock
                return 0,[]
        else
                return currentProfit,[]
```

Case 1: You do not have a stock.

a.Choose to skip buying the stock.

This happend when stockOptions == stockPrices.length we move a day ahead hence day+1 , we keep the stock with us hence it remains haveStock , currentProfit remains same

b.Choose to buy the stock

We have options for stockOptions from i = 0 to stockPrices.length-1 we move a day ahead hence day+1 , we buy so haveStock becomes true and we subtract stockPrices[i][day] from the current profit

```
OPT(day,k,haveStock,stockOptions,currentProfit )
=
max{
   OPT(day+1,k,haveStock,i,currentProfit)                    if  stockOptions == stockPrices.length
    OPT(day+1,k,true,i,currentProfit- stockPrices[i][day])     for 0<=  i <= stockPrices.length-1
}
```

Case 2: You have a stock

a.Choose to skip selling the stock

You move a day ahead hence day + 1; haveStock remains as it is; stockOptions remains as it is ; currentProfit doesnt change

b.Choose to sell the stock

You move a day ahead hence day +1, you complete a transaction hence k-1; haveStock remains as it is; stockOptions remains as it is ;currentProfit adds stockPrices[stockOption][day]

```
OPT(day,k,haveStock,stockOptions,currentProfit )
=
max{
   OPT(day + 1,k,haveStock,stockOptions,currentProfit )
    OPT(day,k-1, ! haveStock,stockOptions,currentProfit + stockPrices[stockOption][day])
}
```

Pseudo Code:

```
Fn(stockPrices,day,k,haveStock,stockOptions,currentProfit):
if k = 0 OR day = stockPrices[0].length
        if day == stockPrices[0] and haveStock
                return 0,[]
         else
                return currentProfit,[]
if(!haveStock)
        for(i = 0 to stockPrices.length)
                if(i == stockPrices.length)
                         Fn(stockPrices,day+1,k,haveStock,i,currentProfit)
                        return max profit and transaction returned by the function
                else
                         Fn(stockPrices,day+1,k,true,i,currentProfit- stockProfit[i][day])
                       return max profit and transaction returned by the function
else
        Fn(stockPrices,day,k-
1,!haveStock,stockOptions,currentProfit+stockProfit[stockOpti][day]
        Fn(stockPrices,day+1,k,haveStock,stockOptions,currentProfit)
        return max profit and transaction returned by the function
```

b.CORRECTNESS

Proof by induction:

OPT(day,k,haveStock,stockOptions,currentProfit) can be proven true if we prove that all the choices we make provide the correct result individually and the choices we make cover all the possible combinations.
On each day we check whether we have a stock or do not have a stock.
a.If we do not have a stock with us we are present in the buy mode. So we have the choice to either buy the stock or skip buying the stock. If we choose to buy the stock we have the choice of buying stocks 1 to m. If we choose to skip buying we move to the next day.
b.If we have stock with us we are in sell mode. Now you can choose to either sell the stock or skip selling the stock. If we sell the stock we calculate the profit from the previous buy. If we skip selling we move a day ahead.
So here we try all possible combinations and make k transactions and then find the maximum transactions. Hence we can conclude that we have the correct result at the end.

Proof of termination:
The algorithm runs till base case. At the base case we will have either k= 0 or day = stockPrices[0].length
As we reduce k and increase day at the recursive calls. At the base case we start returning the values and the method calls will pop from the stack resulting in termination.

c.TIME COMPLEXITY
Time: O(m*n^2k)

We iterate over stocks of size m which gives time complexity of O(m).
In each iteration we call the recursive function. Each recursive function calls n functions and this is done till the depth of 2k. So the recursion tree looks like each node calling n nodes with height 2k hence it has complexity of O(m*n^2k)

d. SPACE COMPLEXITY
Space Complexity :O(1)
Recursive Stack Space Complexity :O(m*2k)

We do not use any data structure to store our results hence it is O(1)
But recursive calls take stack space so we have m iterations and in each iteraion the recursive calls have a depth of 2k hence the space complexity is O(m*2k)

ALGORITHM 5

a. DESIGN

Defination:

OPT(day,k,haveStock,stockOptions,currentProfit)

k = number of transactions

haveStock = check whether you have a stock or not

stockOptions = options if taking any stockk for a given day or not taking a stock at all

currentProfit = profit until now

Algorithm:

Base Case:

```
if k = 0 OR day = stockPrices[0].length
        if day == stockPrices[0] and haveStock
                return 0,[]
         else
                return currentProfit,[]
```

Case 1: You do not have a stock.

a.Choose to skip buying the stock.

This happend when stockOptions == stockPrices.length we move a day ahead hence day+1 , we keep the stock with us hence it remains haveStock , currentProfit remains same

b.Choose to buy the stock

We have options for stockOptions from i = 0 to stockPrices.length-1 we move a day ahead hence day+1 , we buy so haveStock becomes true and we subtract stockPrices[i][day] from the current profit

```
OPT(day,k,haveStock,stockOptions,currentProfit )
=
max{
   OPT(day+1,k,haveStock,i,currentProfit)                    if  stockOptions == stockPrices.length
    OPT(day+1,k,true,i,currentProfit- stockPrices[i][day])     for 0<=  i <= stockPrices.length-1
}
```

Case 2: You have a stock

a.Choose to skip selling the stock

You move a day ahead hence day + 1; haveStock remains as it is; stockOptions remains as it is ; currentProfit doesnt change

b.Choose to sel the stock
You move a day ahead hence day +1, you complete a transaction hence k-1; haveStock remains as it is; stockOptions remains as it is ;currentProfit adds stockPrices[stockOption][day]

```
OPT(day,k,haveStock,stockOptions,currentProfit )
=
max{
   OPT(day + 1,k,haveStock,stockOptions,currentProfit )
    OPT(day,k-1, ! haveStock,stockOptions,currentProfit + stockPrices[stockOption][day])
}
```

Now we memoize the code by storing the values in a map and avoiding the overlapping subproblems as shown in the pseudo code

Pseudo Code*:*

```
Func(stockPrices,day,k,haveStock,stockOptions,currentProfit):

if k = 0 OR day = stockPrices[0].length
        if day == stockPrices[0] and haveStock
                return 0,[]
         else
                return currentProfit,[]
if(haveStock)                                                   ..Memoization
        if map contains key “day@haveStock@stockOptions@k”
        find max profit  and transaction from the map and add currentProfit to it
else
        if map contains key “day@haveStock@k”
        find max profit and transaction from the mapand add currentProfit to it

if(!haveStock)
        for(i = 0 to stockPrices.length)
                if(i == stockPrices.length)
                        Func(stockPrices,day+1,k,haveStock,i,currentProfit)
                        return max profit and transaction returned by the function
                else
```

Func(stockPrices,day+1,k,true,i,currentProfit- stockProfit[i][day])
return max profit and transaction returned by the function
In map put the key "day@haveStock@k" and add profitTillNow-currentProfit and path

else

Func(stockPrices,day+1,k,haveStock,stockOpt,currentProfit)
Func(stockPrices,day,k-1,!haveStock,stockOpt,currentProfit+stockProfit[stockOpti][day])
return max profit and transaction returned by the function
In map put key "day@haveStock@stockOption@k" and add profitTillNow-currentProfit

b. CORRECTNESS

Proof by induction:

OPT(day,k,haveStock,stockOptions,currentProfit) can be proven true if we prove that all the choices we make provide the correct result individually and the choices we make cover all the possible combinations.

On each day we check whether we have a stock or do not have a stock.

a.If we do not have a stock with us we are present in the buy mode. So we have the choice to either buy the stock or skip buying the stock. If we choose to buy the stock we have the choice of buying stocks 1 to m. If we choose to skip buying we move to the next day.

b.If we have stock with us we are in sell mode. Now you can choose to either sell the stock or skip selling the stock. If we sell the stock we calculate the profit from the previous buy. If we skip selling we move a day ahead.

So here we try all possible combinations and make k transactions and then find the maximum transactions. Hence we can conclude that we have the correct result at the end.

Also we use maps to avoid overlapping subproblems but they do not change the choices we make. We just avoid making the same combination of choices by storing the previous available results with us

Proof of termination:

The algorithm runs till base case. At the base case we will have either k= 0 or day = stockPrices[0].length

As we reduce k and increase day at the recursive calls. At the base case we start returning the values and the method calls will pop from the stack resulting in termination.

c.TIME COMPLEXITY

Time: O(m*n^2*k)

We call the recursive function for all the available choices. But we also store the results in a map so that we can avoid the overlapping subproblem issue. As a result we make only n^2*k calls resulting on a time complexity of O(m*n^2*k)

d. SPACE COMPLEXITY

Space Complexity :O(m*n*k)
Recursive Stack Space Complexity :O(m*2k)

We use a map for storing the results. So the space complexity will be the number of keys used in maps. We store days(n), stockOptions(m) , transactions(k) and haveStock(2 choices) so we have m*n*k*2 keys as 2 is a constant we ignore it resulting in space complexity of O(m*n*k)
Recursive calls take stack space so we have m iterations and in each iteraion the recursive calls have a depth of 2k hence the space complexity is O(m*2k)

ALGORITHM 6A

a.DESIGN

Defination:

OPT(i,sellPosition,m,n,k)

m = number of stocks

n = number of days

i = current day

k = number of transactions

sellPosition = boolean which tells whether we can sell or not

dpBuy – 3d array which stores the values of buy

dpSell –3d array which stores values of sell

dpBuyTransaction – 3d array which stores pair of buy transactions

dpSellTransaction –3d array which stores pair of sell transactions

stockPrices –m xn matrix with m stocks for n days

Algorithm:

Case 1: You are in sell position

a.You can choose not sell the stock so you move ahead in day ie i+1 and sellPosition remains true and since you dont make a transaction k remains same

b.You can choose to sell the stock present with us hence you dont move a day ahead, sell position is false and k reduces by 1 as you make a transaction

```
OPT(i,sellPosition,m,n,k )
=
max{
        OPT(i+1,true,m,n,k )
        OPT(i,false,m,n,k-1)
}
```

Case 2: You are not in sell position

You can make choices from m stocks present with us and choose which stock to take on that day or you can skip buying the stock hence we loop for all the m choices and make the choice of buying or not buying the stock

```
OPT(i,sellPosition,m,n,k )
=
max{
```

OPT(i+1,false,mi,n,k) for 1<= mi <= m
OPT(i+1,true,mi,n,k) for 1<= mi <= m
}

We can now memoize the code by storing the overlapping subproblems in the dp arrays as shown in pseudo code below

Pseudo Code:

```
Function(i,sellPosition,m,n,k):

If in sellPosition                                          ..Memoization
        If dpSell[m][k][i] != -1
            maxProfit  = dpSell[m][k][i]
            maxTransaction = dpSellTransaction[m][k][i]
            return maxProfit,maxTransaction
If not in sell position
        If dpBuy[m][k][i] != -1
            maxProfit  = dpBuy[m][k][i]
            maxTransaction = dpBuyTransaction[m][k][i]
            return maxProfit,maxTransaction

if k == 0
        maxProfit = 0
        maxTransactions = []
        return maxProfit,maxTransaction

if i == n
        maxProfit = 0
        maxTransactions = []
        return maxProfit,maxTransaction

if i == n
        if in sellPosition
                maxProfit = stockPrices[m][i]
                maxTransactions = [m,-1,i,stockPrices[m][i]]
                return maxProfit,maxTransaction
        If not in sell position
                maxProfit = 0
                maxTransactions = []
```

```
                return maxProfit,maxTransaction

If in sellPosition
        val1,transaction1 = Function(i+1,true,m,n,k)
        val2,transaction2 = Function(i,false,m,n,k-1)
        dpBuy[m][k-1][i] = val1
        dpSell[m][k][i+1] = val2
        dpSellTransaction[m][k][i-1] = transaction1
        dpSellTransaction[m][k][i+1] = transaction2

        if val1 > val2  + stockPrices[m][i]
                maxProfit = val1
                maxTransaction = transaction1 + [m,-1,i,stockPrices[m][i]]
                return maxProfit ,maxTransaction
        maxProfit = val2
        maxTransaction = transaction2 + [m,-1,i,stockPrices[m][i]]
        return maxProfit ,maxTransaction

else
        totalMax initialise to 0
        for(mi = 0 to m)
                val1,transaction1 = Function(i+1,false,mi,n,k)
                val2,transaction2 = Function(i+1,true,mi,n,k-1)
                dpBuy[mi][k][i+1] = val1
                dpSell[mi][k][i+1] = val2
                dpSellTransaction[mi][k][i+1] = transaction1
                dpSellTransaction[mi][k][i+1] = transaction2

                val2 = val1 - stockPrices[m][i]
                forward1 = []; forward2 = []
                for(tE in transaction2)
                        if(tE[1] == -1)
                                forward2.add( tE[0],i,tE[2],tE[3] - stockProces[mi][i])
                        else
                                forward1.add(tE)
                Sort forward2 based on 3rd index

                tempMax = max(val1,val2)
```

```
            If totalMax< tempMax
                if val1 == temp
                    transaction3 = transaction1
                else
                    transaction3 = forward1 + forwar2[0]
                totalMax = tempMax
return totalMax,transaction3
```

b.CORRECTNESS

Proof by induction:

OPT(i,sellPosition,m,n,k) can be proven true if we prove that all the choices we make provide the correct result individually and the choices we make cover all the possible combinations.

On each day we check whether we have a stock or do not have a stock.

a.If we do not have a stock with us we are present in the buy mode. So we have the choice to either buy the stock or skip buying the stock. If we choose to buy the stock we have the choice of buying stocks 1 to m. If we choose to skip buying we move to the next day.

b.If we have stock with us we are in sell mode. Now you can choose to either sell the stock or skip selling the stock. If we sell the stock we calculate the profit from the previous buy. If we skip selling we move a day ahead.

So here we try all possible combinations and make k transactions and then find the maximum transactions. Hence we can conclude that we have the correct result at the end.

Also we use dp to avoid overlapping subproblems but they do not change the choices we make. We just avoid making the same combination of choices by storing the previous available results with us. The dp array will store the max profit and maxTransactiions in them for buy sell cases. We can use the dp values to get the appropriate result

Proof of termination:

The algorithm runs till base case. At the base case we will have either k= 0 or i=n

As we reduce k and increase day at the recursive calls. At the base case we start returning the values and the method calls will pop from the stack resulting in termination.

c.TIME COMPLEXITY

Time: O(m*n*k)

We call the recursive function for all the available choices. But we also store the results in a dp so that we can avoid the overlapping subproblem issue. As a result we make only n*k calls which is less in comparison to the calls in above algorithm as we used improved memoization in out resulting on a time complexity of O(m*n*k)

d. SPACE COMPLEXITY

Space Complexity :O(m*n*k)

Recursive Stack Space Complexity :O(m*2k)

We use a dp for storing the results. So the space complexity will be the size of the dp used. We store days(n), stockOptions(m) , transactions(k) and haveStock(2 choices) so we have m*n*k*2 array size and as 2 is a constant we ignore it resulting in space complexity of O(m*n*k)
Recursive calls take stack space so we have m iterations and in each iteraion the recursive calls have a depth of 2k hence the space complexity is O(m*2k)

ALGORITHM 6B

a. DESIGN

Def:

stockPrices = mxn matrix with m stocks for n days
noOf Stocks = n
noOfDays = m

Pseudo Code:

```
if(noOfDays == 0){
   return [-1,-1,-1]

Initialise result array of size k+1* noOfDays result
maxProfitTillNow = 0;

for(i = 1 to k ) {
        for(j = 0 to noOfStocks ) {
                maximumProfitTillNow= -INF
                for(d= 1 to noOfDays) {
                  maxProfitTillNow =max(maxProfitTillNow ,result[i-1][d-1] stockPrices[j][d-1]);
                  result[i][d] = max(result[i][d-1],result[i][d],stockProces[j][d] +maxProfitTillNow

backtracking to get the potential stocks and their dates when bought and sold
currentDay = noOfDays-1;
transactionsUsed = k;
buyDay = currentDay-1;
sellDay = currentDay;
found = false;
while(transactionsUsed != 0 and sellDay > 0 and buyDay  != -1){
        found = false;
        if(result[transactionsUsed][sellDay] == result[transactionsUsed][sellDay-1]) {
                Move sellDay ahead ;
                Move buyDay behinf;
        else if(result[transactionsUsed][sellDay] > result[transactionsUsed-1][sellDay-1])
                value = result[transactionsUsed][sellDay] - result[transactionsUsed - 1][buyDay];
                if(transactionsUsed - 1 == 0 && result[transactionsUsed-1][buyDay] == 0)
                        Find the equivalent stocks for the day
                        Break out of loop
                else
                        while we do not find  {
                        value=result[transactionsUsed][sellDay]
                                -result[transactionsUsed-1][buyDay];
```

```
                    found= find if the equivalent stocks are found yetr
                     if(!found)
                            Move buyDay behind
                    transactionsUsed = transactionsUsed-1;
                    sellDay = buyDay;
                    buyDay = sellDay - 1;
return outputList
```

b.CORRECTNESS

Since the algorithm uses loops to find the answer we can use loop invariant stratergy to prove the correctness of the algorithm

Proof by Loop Invariant:

Since the algorithm uses loops to find the answer we can use loop invariant stratergy to prove the correctness of the algorithm

Initialization

We initialise each iteration of i,j and d representing transactions stocks and days with 1
We initialize maxProfitTillNow as 0 befor the loop begins.We initialise result arraywhich will store our result values at any given instance So these values are correct before the loop begins

Loop Invariant

The loop iterates from i = 1 to k and j = 1 to m and d from 1 to n. maxProfitTillNow holds the maxProfit of the current stocks resultholds the maximum profit until now.We will have maximum profit transaction for all the stocks when the outer loop ends.

Mantainance:

We iterate through transactions,stocks,days and find the maximum of the values of previous result - stockPrices of previous day ie result[i-1][d-1] -stockPrices[j][d-1]. We will have max profit for a single by checking with finalMaxProfit . So if this iteration holds correct values then all iterations will hold correct values. Hence the correct values are maintained in each loop iteration

Termination:

The outer loop terminates when i reaches k. At the end of the loop we have the maximum profitable transaction of all the stocks and will be present at the last index of the dp array.So the values are correct when loop ends.

c.TIME COMPLEXITY

Time: O(m*n*k)

As we use tabulation technique we use 3 for m , n , k and store the results in an dp array. So the total time complexity will be O(m*n*k)

d. SPACE COMPLEXITY

Space Complexity :O(m*n*k)

We use a 3dimensional dp array to store result with size m*n*k hence the space complexity

ALGORITHM 7

a. DESIGN

Definition:

OPT(day,haveStock,stockOptions,currentProfit,c)

c = cooldown period

haveStock = check whether you have a stock or not

stockOptions = options if taking any stockk for a given day or not taking a stock at all
currentProfit = profit until now

Algorithm:

Base Case:

```
if day > stockPrices[0].length
        If  haveStock
                return 0,[]
         else
                return currentProfit,[]
```

Case 1: You do not have a stock.

a.Choose to skip buying the stock.

This happend when stockOptions == stockPrices.length we move a day ahead hence day+1 , we keep the stock with us hence it remains haveStock , currentProfit remains same , c remaoins

b.Choose to buy the stock

We have options for stockOptions from i = 1 to stockPrices.length we move a day ahead hence day+1 , we buy so haveStock becomes true and we subtract stockPrices[i][day] from the current profit

```
OPT(day,k,haveStock,stockOptions,currentProfit,c )
=
max{
   OPT(day+1,haveStock,i,currentProfit,c)                    if  stockOptions == stockPrices.length
    OPT(day+1,true,i,currentProfit- stockPrices[i][day],c)   for 1<=  i <= stockPrices.length-1
}
```

Case 2: You have a stock

a.Choose to skip selling the stock

You move day ahead by day +1; haveStock remains as it is; stockOptions does not change and remains as it is ; currentProfit doesnt change

b.Choose to sel the stock

You move a day ahead by day +c+1 to allow for cooldown,you complete a transaction hence k changes to k-1; haveStock remains as it is; stockOptions remains as it is ;currentProfit changes as we sell and we add stockPrices[stockOption][day] to currentProfit

```
OPT(day,haveStock,stockOptions,currentProfit,c )
=
max{
   OPT(day +1,haveStock,stockOptions,currentProfit,c )
    OPT(day+c+1, ! haveStock,stockOptions,currentProfit + stockPrices[stockOption][day],c)
}
```

Pseudo Code:

```
Func(stockPrices,day,haveStock,stockOptions,currentProfit,c):
if day > stockPrices[0].length
        if haveStock
                return 0,[]
         else
                return currentProfit,[]
if(!haveStock)
        for(i = 0 to stockPrices.length)
                if(i == stockPrices.length)
                        Func(stockPrices,day+1,haveStock,i,currentProfit,c)
                        return max profit and transaction returned by the function
                else
                       Func(stockPrices,day+1,k,true,i,currentProfit- stockProfit[i][day],c)
                        return max profit and transaction returned by the function
else
        Func(stockPrices,day+1,haveStock,stockOptions,currentProfit,c)
        Func(stockPrices,day+c+1,!haveStock,stockOptions,currentProfit +
              stockPrices[stockOption][day] ,c)
        return max profit and transaction returned by the function
```

b.CORRECTNESS

Proof by induction:

OPT(day,haveStock,stockOptions,currentProfit,c) can be proven true if we prove that all the choices we make provide the correct result individually and the choices we make cover all the possible combinations.
On each day we check whether we have a stock or do not have a stock.
a.If we do not have a stock with us we are present in the buy mode. So we have the choice to either buy the stock or skip buying the stock. If we choose to buy the stock we have the choice of buying stocks 1 to m and move to the next day. If we choose to skip buying we move to the next day.
b.If we have stock with us we are in sell mode. Now you can choose to either sell the stock or skip selling the stock. If we sell the stock we calculate the profit from the previous buy and move to day+c+1 to allow for cooldown. If we skip selling we move a day ahead.
So here we try all possible combinations and make many transactions with a cooldown period if we buy and then find the maximum transactions. Hence we can conclude that we have the correct result at the end.

Proof of termination:
The algorithm runs till base case. At the base case we will have day >stockPrices[0].length
As we reduce day at the recursive calls. At the base case we start returning the values and the method calls will pop from the stack resulting in termination.

c.TIME COMPLEXITY
Time: O(m*2^n)
We iterate over stocks of size m which gives time complexity of O(m).
In each iteration we call the recursive function. Each recursive function calls 2 functions and this is done till the depth of n. So the recursion tree looks like each node calling 2 nodes with height n hence it has complexity of O(m*2^n)

d. SPACE COMPLEXITY
Space Complexity :O(1)
Recursive Stack Space Complexity :O(m*n)
We do not use any data structure to store our results hence it is O(1)
But recursive calls take stack space so we have m iterations and in each iteraion the recursive calls have a depth of n hence the space complexity is O(m*n)

ALGORITHM 8

a.DESIGN
Definition:

OPT(day,haveStock,stockOptions,currentProfit,c)
c = cooldown period
haveStock = check whether you have a stock or not
stockOptions = options if taking any stockk for a given day or not taking a stock at all
currentProfit = profit until now

Algorithm:
Base Case:
if day > stockPrices[0].length
    If haveStock
        return 0,[]
    else
        return currentProfit,[]

Case 1: You do not have a stock.
a.Choose to skip buying the stock.
This happend when stockOptions == stockPrices.length we move a day ahead hence day+1 , we keep the stock with us hence it remains haveStock , currentProfit remains same , c remaoins

b.Choose to buy the stock
We have options for stockOptions from i = 1 to stockPrices.length we move a day ahead hence day+1 , we buy so haveStock becomes true and we subtract stockPrices[i][day] from the current profit

OPT(day,k,haveStock,stockOptions,currentProfit,c)
=
max{
  OPT(day+1,haveStock,i,currentProfit,c) if stockOptions == stockPrices.length
  OPT(day+1,true,i,currentProfit- stockPrices[i][day],c) for 1<= i <= stockPrices.length-1
}

Case 2: You have a stock

a.Choose to skip selling the stock
You move day ahead by day +1; haveStock remains as it is; stockOptions does not change and remains as it is ; currentProfit doesnt change

b.Choose to sel the stock

You move a day ahead by day +c+1 to allow for cooldown,you complete a transaction hence k changes to k-1; haveStock remains as it is; stockOptions remains as it is ;currentProfit changes as we sell and we add stockPrices[stockOption][day] to currentProfit

```
OPT(day,haveStock,stockOptions,currentProfit,c )
=
max{
   OPT(day +1,haveStock,stockOptions,currentProfit,c )
    OPT(day+c+1, ! haveStock,stockOptions,currentProfit + stockPrices[stockOption][day],c)
}
```

Now we memoize the code by storing the values in a map and avoiding the overlapping subproblems as shown in the pseudo code

Pseudo Code*:*

```
Func(stockPrices,day,haveStock,stockOptions,currentProfit,c):

if day > stockPrices[0].length
        if haveStock
                return 0,[]
         else
                return currentProfit,[]
if(haveStock)                                                   ..Memoization
        if map contains key “day@haveStock@stockOptions@”
        find max profit  and transaction from the map and add currentProfit to it
else
        if map contains key “day@haveStock@”
        find max profit and transaction from the mapand add currentProfit to it

if(!haveStock)
        for(i = 1 to stockPrices.length)
                if(i == stockPrices.length)
                        Func(stockPrices,day+1,haveStock,i,currentProfit,c)
                       return max profit and transaction returned by the function
                else
                        Func(stockPrices,day+1,k,true,i,currentProfit- stockProfit[i][day])
                      return max profit and transaction returned by the function
```

In map put the key "day@haveStock@" and add profitTillNow-currentProfit and path

Else

Func(stockPrices,day+1,k,haveStock,stockOpt,currentProfit,c)

Func(stockPrices,day+c+1,haveStock,stockOption,

currentProfit+stockProfit[stockOpti][day],c)

return max profit and transaction returned by the function

In map put key "day@haveStock@stockOption" and add profitTillNow-currentProfit

b.CORRECTNESS

Proof by induction:

OPT(day,k,haveStock,stockOptions,currentProfit) can be proven true if we prove that all the choices we make provide the correct result individually and the choices we make cover all the possible combinations.

On each day we check whether we have a stock or do not have a stock.

a.If we do not have a stock with us we are present in the buy mode. So we have the choice to either buy the stock or skip buying the stock. If we choose to buy the stock we have the choice of buying stocks 1 to m. If we choose to skip buying we move to the next day.

b.If we have stock with us we are in sell mode. Now you can choose to either sell the stock or skip selling the stock. If we sell the stock we calculate the profit from the previous buy. If we skip selling we move a day ahead.

So here we try all possible combinations and make k transactions and then find the maximum transactions. Hence we can conclude that we have the correct result at the end.

Also we use map to avoid overlapping subproblems but they do not change the choices we make. We just avoid making the same combination of choices by storing the previous available results with us

Proof of termination:

The algorithm runs till base case. At the base case we will have either k= 0 or day = stockPrices[0].length

As we reduce k and increase day at the recursive calls. At the base case we start returning the values and the method calls will pop from the stack resulting in termination.

c.TIME COMPLEXITY

Time: O(m*n^2*k)

We call the recursive function for all the available choices. But we also store the results in a map so that we can avoid the overlapping subproblem issue. As a result we make only n^2*k calls resulting on a time complexity of O(m*n^2*k)

d. SPACE COMPLEXITY

Space Complexity :O(m*n*k)
Recursive Stack Space Complexity :O(m*2k)

We use a map for storing the results. So the space complexity will be the number of keys used in maps. We store days(n), stockOptions(m) , transactions(k) and haveStock(2 choices) so we have m*n*k*2 keys as 2 is a constant we ignore it resulting in space complexity of O(m*n*k)
Recursive calls take stack space so we have m iterations and in each iteraion the recursive calls have a depth of 2k hence the space complexity is O(m*2k)

ALGORITHM 9A

a.DESIGN

Defination:

OPT(i,sellPosition,m,n,c)

m = number of stocks

n = number of days

i = current day

c =cool down peropd

sellPosition = boolean which tells whether we can sell or not

dpBuy – 2d array which stores the values of buy

dpSell –2array which stores values of sell

dpBuyTransaction – 2d array which stores pair of buy transactions

dpSellTransaction –2d array which stores pair of sell transactions

stockPrices –m xn matrix with m stocks for n days

Algorithm:

Case 1: You are in sell position

a.You can choose not sell the stock so you move ahead in day ie i+1 and sellPosition remains true

b.You can choose to sell the stock present with us hence you move a day ahead by day + c + 1, sell position is false

```
OPT(i,sellPosition,m,n,c )
=
max{
        OPT(i+1,true,m,n,c )
        OPT(i+c+1,false,m,n,c)
}
```

Case 2: You are not in sell position

You can make choices from m stocks present with us and choose which stock to take on that day or you can skip buying the stock hence we loop for all the m choices and make the choice of buying or not buying the stock

```
OPT(i,sellPosition,m,n,c )
=
max{
   OPT(i+1,false,mi,n,c)     for 1<=  mi <= m
    OPT(i+1,true,mi,n,c)     for 1<=  mi <= m
```

}

We can now memoize the code by storing the overlapping subproblems in the dp arrays as shown in pseudo code below

Pseudo Code:

```
Function(i,sellPosition,m,n,c):

If in sellPosition                                          ..Memoization
        If dpSell[m][i] != -1
            maxProfit  = dpSell[m][i]
            maxTransaction = dpSellTransaction[m][i]
            return maxProfit,maxTransaction
If not in sell position
        If dpBuy[m][i] != -1
            maxProfit  = dpBuy[m][i]
            maxTransaction = dpBuyTransaction[m][i]
            return maxProfit,maxTransaction

if k == 0
        maxProfit = 0
        maxTransactions = []
        return maxProfit,maxTransaction

if i == n
        maxProfit = 0
        maxTransactions = []
        return maxProfit,maxTransaction

if i == n
        if in sellPosition
                maxProfit = stockPrices[m][i]
                maxTransactions = [m,-1,i,stockPrices[m][i]]
                return maxProfit,maxTransaction
        If not in sell position
                maxProfit = 0
                maxTransactions = []
                return maxProfit,maxTransaction
```

```
If in sellPosition
        val1,transaction1 = Function(i+1,true,m,n,k)
        val2,transaction2 = Function(i+c+1,false,m,n,k)
        dpBuy[m][i] = val1
        dpSell[m]][i+1] = val2
        dpSellTransaction[m][i-1] = transaction1
        dpSellTransaction[m][i+1] = transaction2

        if val1 > val2  + stockPrices[m][i]
                maxProfit = val1
                maxTransaction = transaction1 + [m,-1,i,stockPrices[m][i]]
                return maxProfit ,maxTransaction
        maxProfit = val2
        maxTransaction = transaction2 + [m,-1,i,stockPrices[m][i]]
        return maxProfit ,maxTransaction

else
        totalMax initialise to 0
        for(mi = 0 to m)
                val1,transaction1 = Function(i+1,false,mi,n,c)
                val2,transaction2 = Function(i+1,true,mi,n,c)
                dpBuy[mi][i+1] = val1
                dpSell[mi][i+1] = val2
                dpSellTransaction[mi][i+1] = transaction1
                dpSellTransaction[mi][i+1] = transaction2

                val2 = val1 - stockPrices[m][i]
                forward1 = []; forward2 = []
                for(tE in transaction2)
                        if(tE[1] == -1)
                                forward2.add( tE[0],i,tE[2],tE[3] - stockProces[mi][i])
                        else
                                forward1.add(tE)
                Sort forward2 based on 3rd index

                tempMax = max(val1,val2)
                If totalMax< tempMax
                        if val1 == temp
```

```
                    transaction3 = transaction1
            else
                    transaction3 = forward1 + forwar2[0]
            totalMax = tempMax
return totalMax,transaction3
```

b.CORRECTNESS

Proof by induction:

OPT(i,sellPosition,m,n,c) can be proven true if we prove that all the choices we make provide the correct result individually and the choices we make cover all the possible combinations.

On each day we check whether we have a stock or do not have a stock.

a.If we do not have a stock with us we are present in the buy mode. So we have the choice to either buy the stock or skip buying the stock. If we choose to buy the stock we have the choice of buying stocks 1 to m. If we choose to skip buying we move to the next day.

b.If we have stock with us we are in sell mode. Now you can choose to either sell the stock or skip selling the stock. If we sell the stock we calculate the profit from the previous buy. If we skip selling we move a day ahead.

So here we try all possible combinations and then find the maximum transactions. Hence we can conclude that we have the correct result at the end.

Also we use dp to avoid overlapping subproblems but they do not change the choices we make. We just avoid making the same combination of choices by storing the previous available results with us. The dp array will store the max profit and maxTransactiions in them for buy sell cases. We can use the dp values to get the appropriate result

Proof of termination:

The algorithm runs till base case. At the base case we will have r i=n

As we increate iy at the recursive calls. At the base case we start returning the values and the method calls will pop from the stack resulting in termination.

c.TIME COMPLEXITY

Time: O(m*n)

We call the recursive function for all the available choices. But we also store the results in a dp so that we can avoid the overlapping subproblem issue. As a result we make only n calls which is less in comparison to the calls in above algorithm as we used improved memoization in out resulting on a time complexity of O(m*n)

d. SPACE COMPLEXITY

Space Complexity :O(m*n)
Recursive Stack Space Complexity :O(m*n)

We use a dp for storing the results. So the space complexity will be the size of the dp used. We store days(n), stockOptions(m) ,and haveStock(2 choices) so we have m*n*2 array size and as 2 is a constant we ignore it resulting in space complexity of O(m*n)
Recursive calls take stack space so we have m iterations and in each iteraion the recursive calls have a depth of n hence the space complexity is O(m*n)

ALGORITHM 9B

a.DESIGN

Def:

stockPrices = mxn matrix with m stocks for n days
noOf Stocks = n
noOfDays = m

Pseudo Code:

```
maximumProfitTillNow=  -INF;

if(noOfDays == 0)
   return [-1,-,1,-1]

Initialise maxProfitBuyStartAt with size noOfDays
for(d = 1 to noOfDays)
   maxProfitBuyStartAt[d] = 0,[]

Intialise pathTillNow as null

for(buy= noOfDays-2 to 1 ){
   for(m=1 to noOfStocks) {
      for (int sell = buy + 1 to noOfDays) {
         if (stockPrices[m][buy] > stockPrices[m][sell])
            continue;

         profit = stockPrices[m][sell] - stockPrices[m][buy];
         if (sell + c + 1 < noOfDays) {
            profit  += maxProfitBuyStartAt[sell + c + 1].profitTillNow
            pathTillNow = [maxProfitBuyStartAt[sell + c + 1].pathTillNow ];
         }

         if (profit > maxProfitBuyStartAt[buy].profitTillNow) {
            maxProfitBuyStartAt[buy].profitTillNow= profit;
            maxProfitBuyStartAt[buy].pathTillNow = clear;
            if (pathTillNow != null and  !pathTillNow is not empty)
               maxProfitBuyStartAt[buy].pathTillNow   + pathTillNow;
            maxProfitBuyStartAt[buy].pathTillNow + [m,sell];
            maxProfitBuyStartAt[buy].pathTillNow + [m,buy];
         }
         if (pathTillNow != null)
            pathTillNow= clear
```

```
    }
    if(maxProfitBuyStartAt[buy + 1].profitTillNow > maxProfitBuyStartAt[buy + 1].profitTillNow)
       maxProfitBuyStartAt[buy + 1].profitTillNow =   maxProfitBuyStartAt[buy].profitTillNow
       maxProfitBuyStartAt[buy].pathTillNow  = clear
       maxProfitBuyStartAt[buy].pathTillNow  += maxProfitBuyStartAt[buy + 1].pathTillNow
    }

return maxProfitBuyStartAt[0];
```

b.CORRECTNESS

Since the algorithm uses loops to find the answer we can use loop invariant stratergy to prove the correctness of the algorithm

Proof by Loop Invariant:

Since the algorithm uses loops to find the answer we can use loop invariant stratergy to prove the correctness of the algorithm

Initialization

We start buy with noOfdays-2,m with 1 and sell with buy + 1. We initialise maxProfitBuyStartAt[] array which stores our result. So these values are correct before the loop begins

Loop Invariant

The loop iterates from buy = noOfDays-2 to 1 and m from 1 to noOfStocks and we sell from buy+1 to noOfDays. maxProfitBuyStartAt will hold the maximum profit and path at the particular index.So we will have correct output when the loop ends

Mantainance:

In each iteration we get maxProfit. We start with sellDay = buyDay +1 and find all the combinations for that stock and store maxProfit in our array. We do the same for next stock and if the profit is highest we reassign the value in the array. So this loop guarantees maxProfit. We can extend the logic that if this loop gives correct output all loops will give correct output.

Termination:

The outer loop terminates when buy reaches 1. At the end of the loop we have the maximum profitable transaction at maxProfitBuyStartAt[0].So the values are correct when the loop ends.

c.TIME COMPLEXITY

Time: O(m*n)

As we use tabulation technique we use a 2d array to store our results and though there are 3 loops we make an exit and conditions making the inner loops linear, s the time complexity is (m*n)

d. SPACE COMPLEXITY

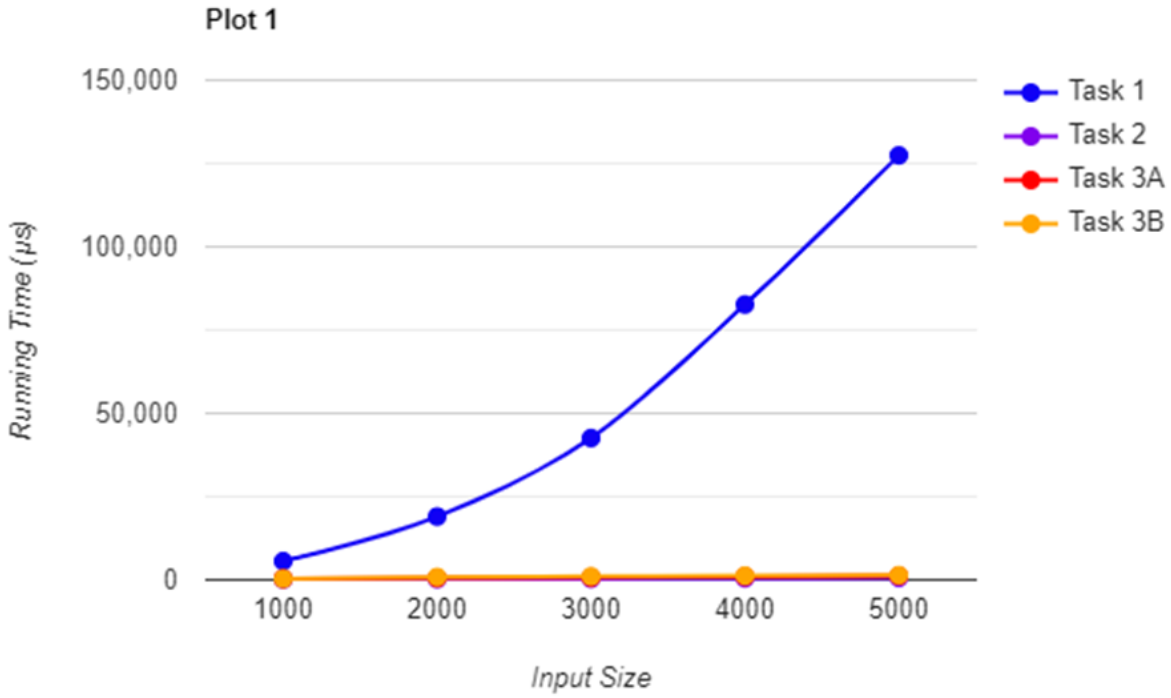
Space Complexity :O(m*n)

We use a 2 dimensional array to store result with size m*n hence the space complexity is O(m*n)

COMPARATIVE STUDY

Plot 1

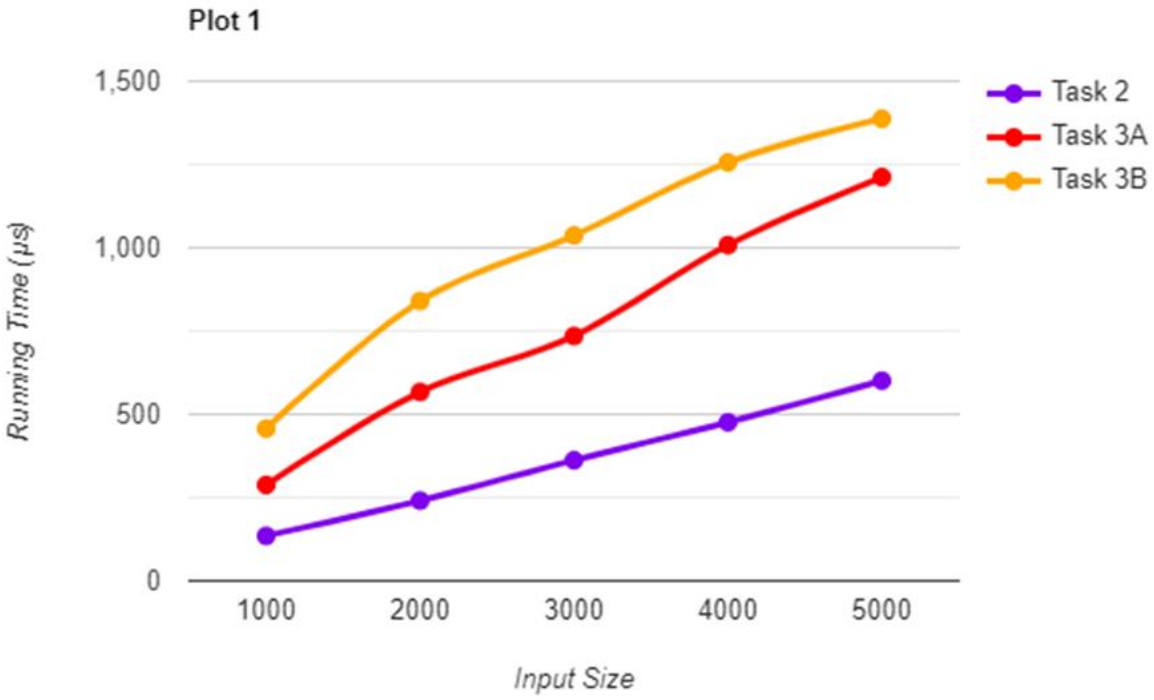
(*Comparison of Task 1, Task 2, Task 3A, Task 3B for variable n and fixed m***)**



For Plot 1, we have executed the Problems using four different solutions with varying time complexities (Running Time). For this Plot, we have kept the number of Stocks (m) fixed and varying the number of days(n) to create 5 different data points for all four approaches.

As observed from the graph above, we can infer that the time taken by brute force approach gradually increases as the input size increases with constant factor tracing the graph of $O(n^2)$.The running time for brute approach increases exponentially when compared to the other approaches. From this, we can incur that as we increase the input time, the brute approach becomes abysmally inefficient.

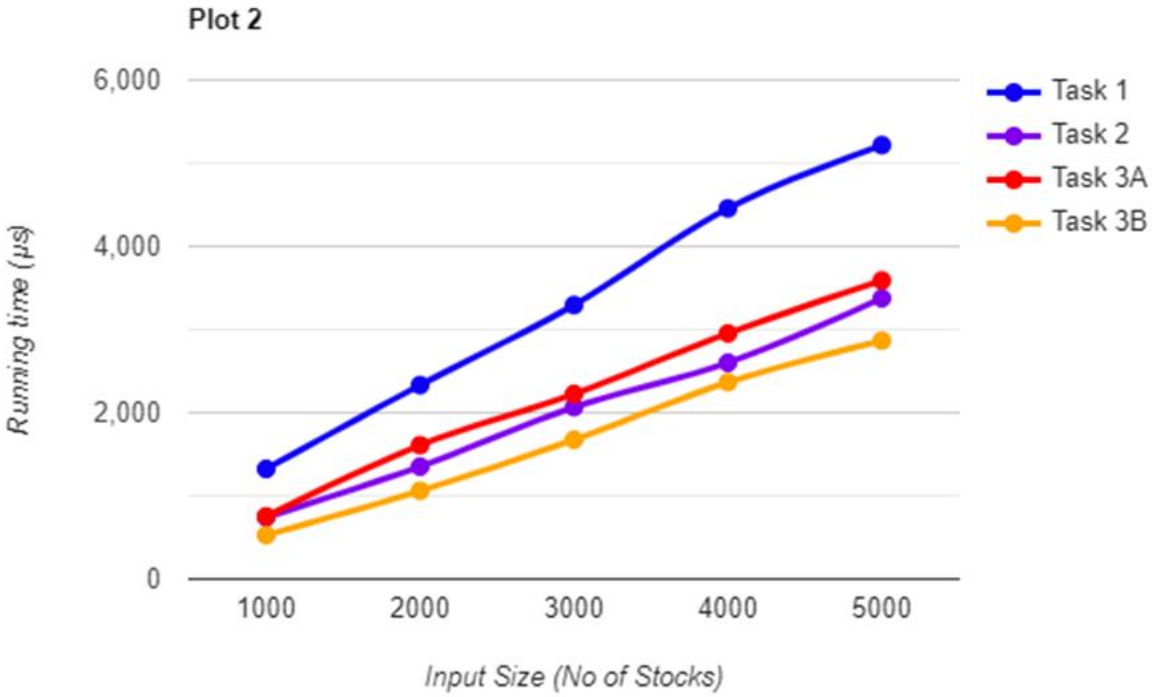
Now, to compare the greedy and dynamic approaches, we will refer to the below graph without the Task 1 line.



Analyzing the graph, we gather that the running time graph lines for Task 2, 3A and 3B traces out the O(n) graph which shows their running time in increasing fashion. Also, we can conclude that in this case for our program, greedy solution fared the best with least running time for varying input of number of days(n).

Plot 2

(*Comparison of Task 1, Task 2, Task 3A, Task 3B for variable m and fixed n***)**

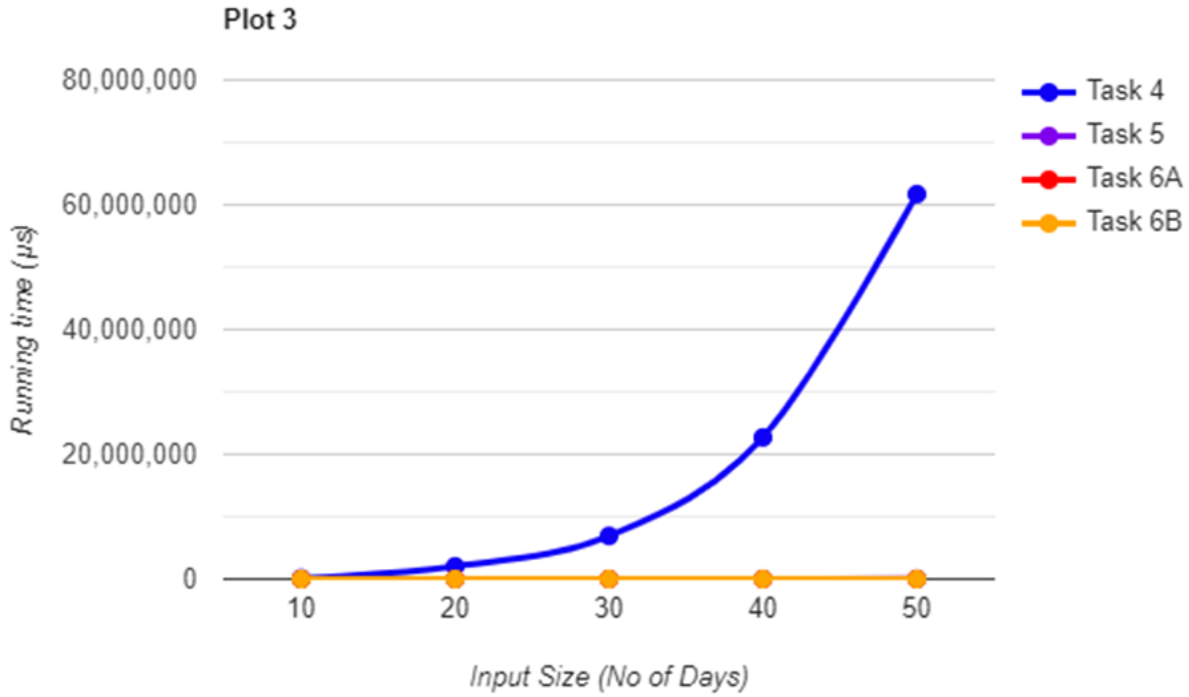


For Plot 2, we have varied the number of stocks(m) and kept the number of days (n) fixed. From the above graph, we can observe that the running times line for all approaches is almost liner. The reason behind this is that we are keeping the time complexity deciding factor n as constant. So When we keep it common them we have m*constant value. Now it will not matter heavily if n is squared or not. It still is a constant.

As seen in the above graph, the line graphs are in increasing order proportional to its constant increase factor.

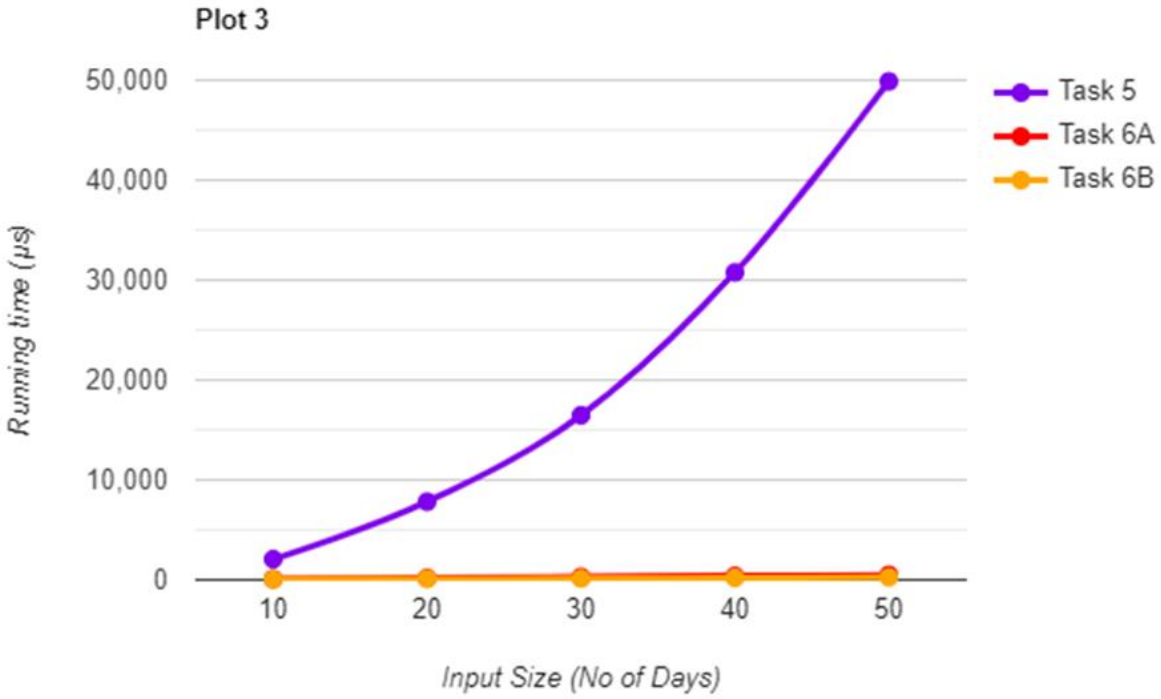
Plot 3

(*Comparison of Task 4, Task 5, Task 6A, Task 6B for variable n, fixed m and fixed k***)**

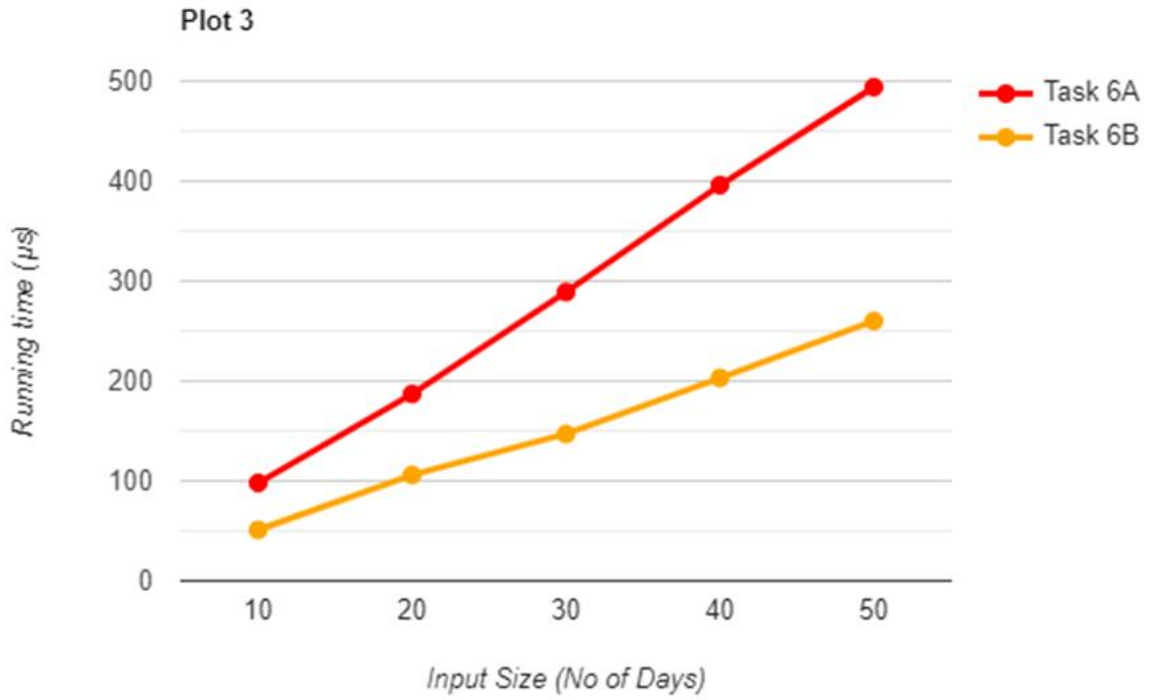


For this plot, we varied the number of days(n) and fixed the number of stocks (m = 10) and number of transactions (k =2). From the above graph, we can deduce that the brute force approach has running time for increasing input increasing with quadratic power. It is tracing the increase of running time in terms of square function.

We will now refer to the graph below without brute running time to analyze the times for comparing other approaches quality.


Plot 3
50,000
40,000
30,000
20,000
10,000
0
Running time (μs)
10
20
30
40
50
Input Size (No of Days)
Task 5
Task 6A
Task 6B

(a)

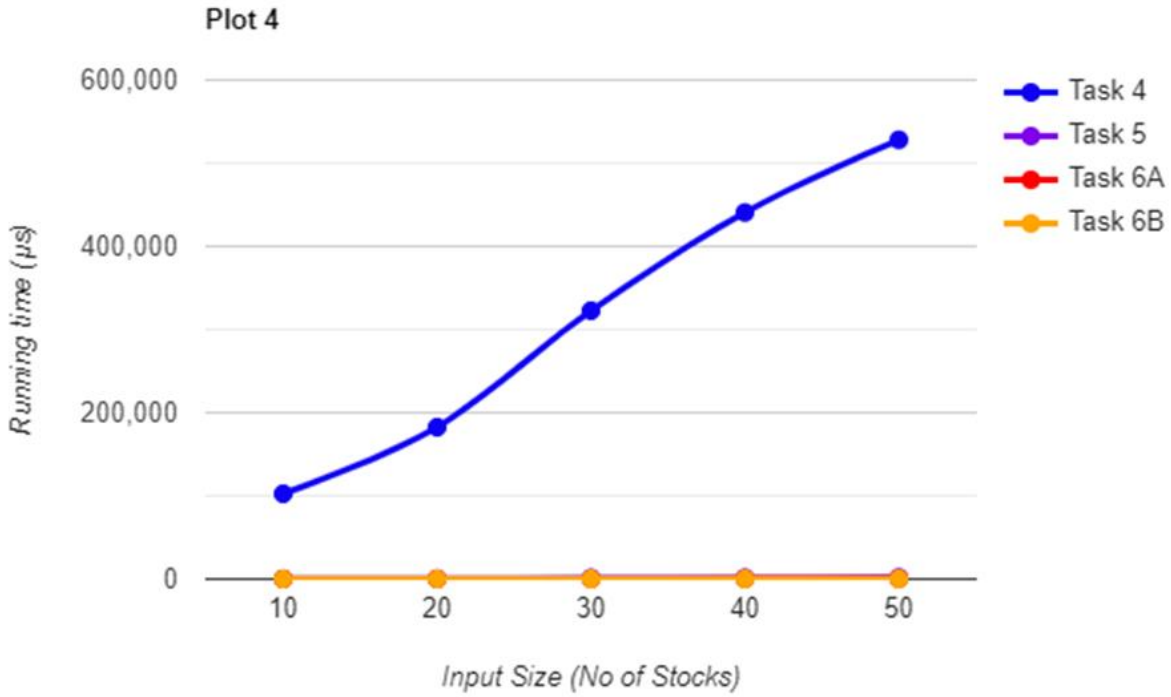


(b)

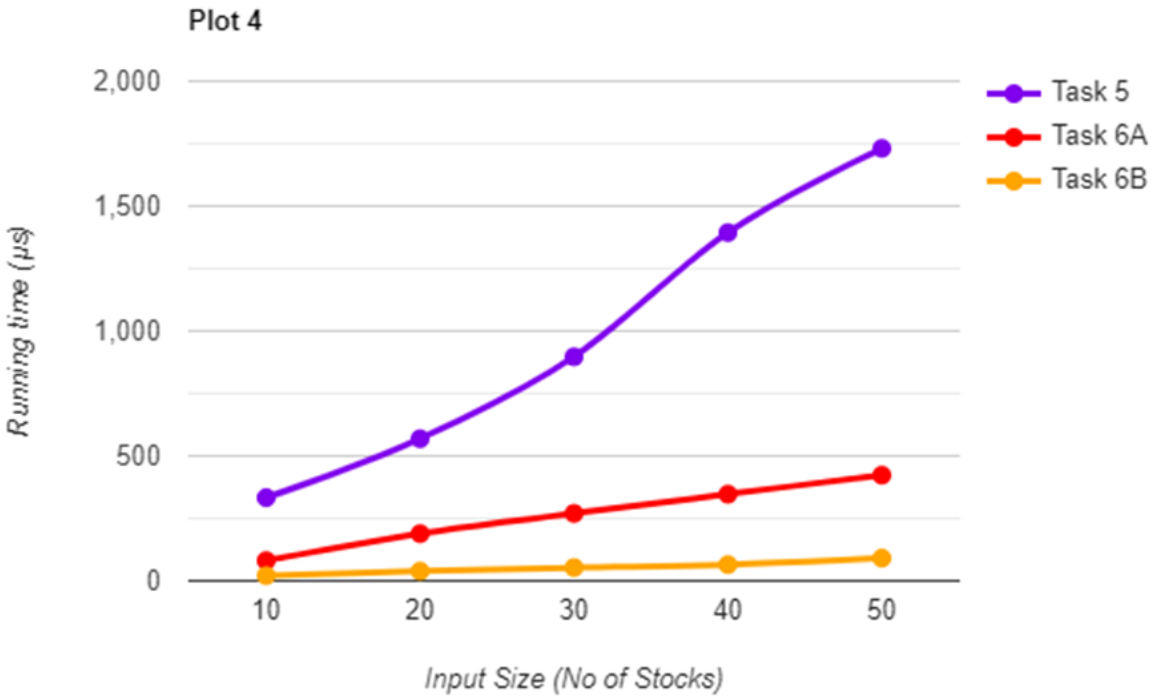
Analyzing both (a) and (b), we can infer that Task 5 traces O(n^2) graph and Task 6A and Task 6B are tracing O(n) graph.

Plot 4

(*Comparison of Task 4, Task 5, Task 6A, Task 6B for variable m, fixed n and fixed k***)**



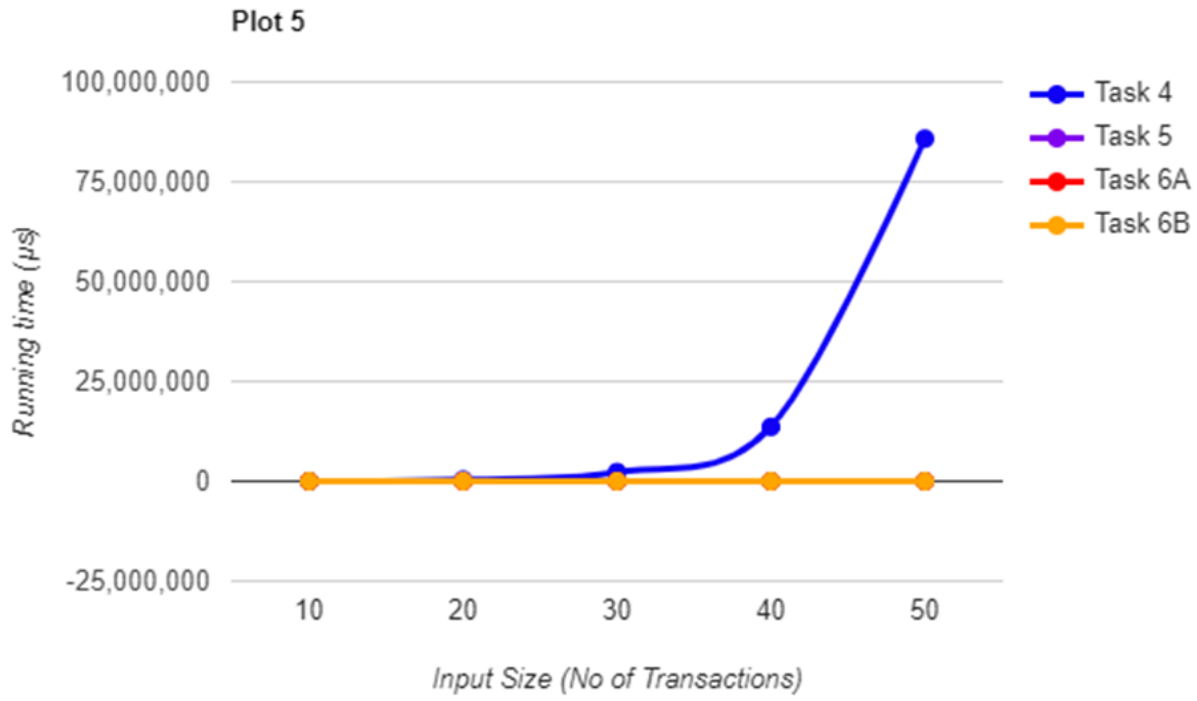
For this plot, we have varied the number of stocks(m) and fixed the number of days (n = 10) and number of transactions (k =2). From the above graph, we can deduce that the brute force approach has running time for increasing input as a linear function. As the input increases, the time will be increase linearly.



As the input size increases with constant factor, we can observe the running time for all tasks are increasing in linear time.

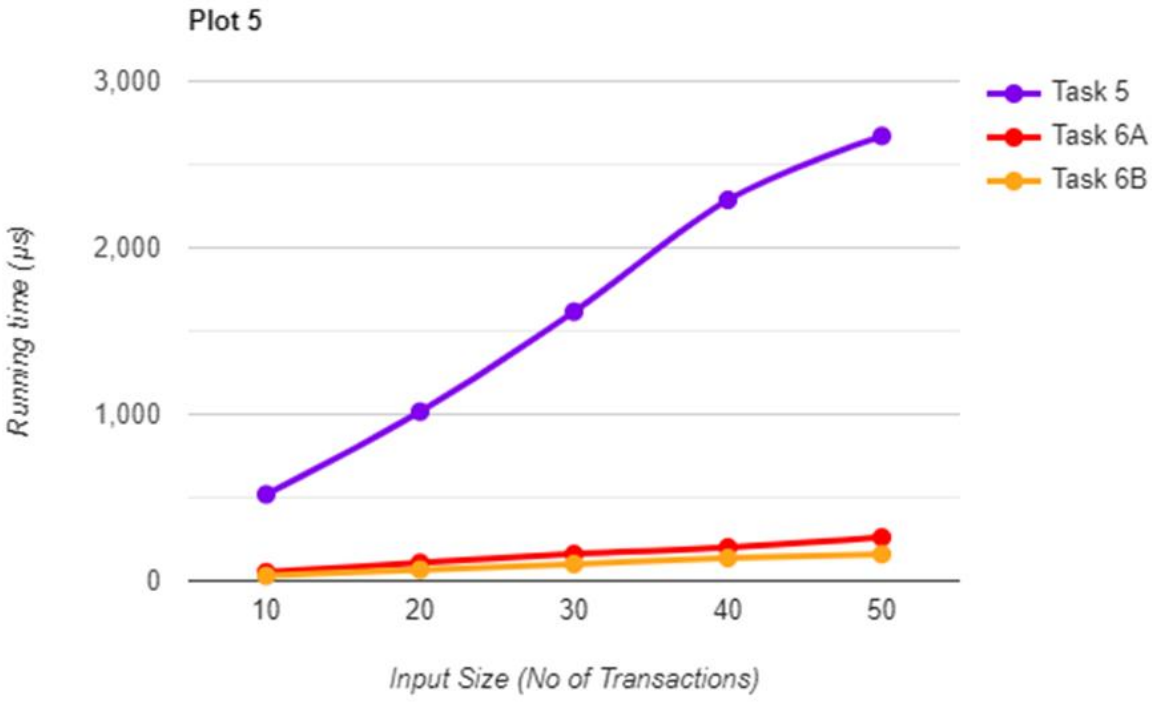
Plot 5

(*Comparison of Task 4, Task 5, Task 6A, Task 6B for variable k, fixed n and fixed m***)**

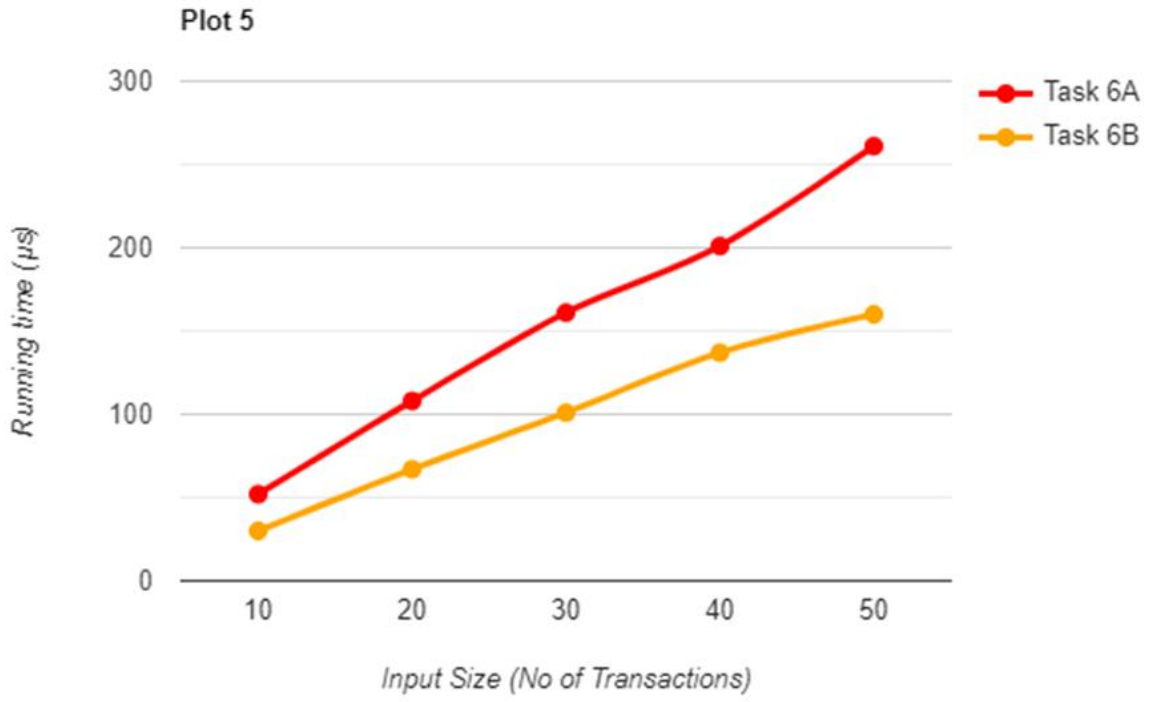


For this plot, we varied the number of transactions(k) and fixed the number of days (n = 8) and number of stocks (k =8). From the above graph, we can deduce that the brute force approach has running time for increasing input increasing with the power of exponential. As the input increases, the brute solution will abysmally slow down.

Let's refer to the below graphs for Task 5, Task6A and Task 6B, to compare the line graphs.


Plot 5
3,000
2,000
1,000
0
Running time (μs)
10
20
30
40
50
Input Size (No of Transactions)
Task 5
Task 6A
Task 6B

(a)

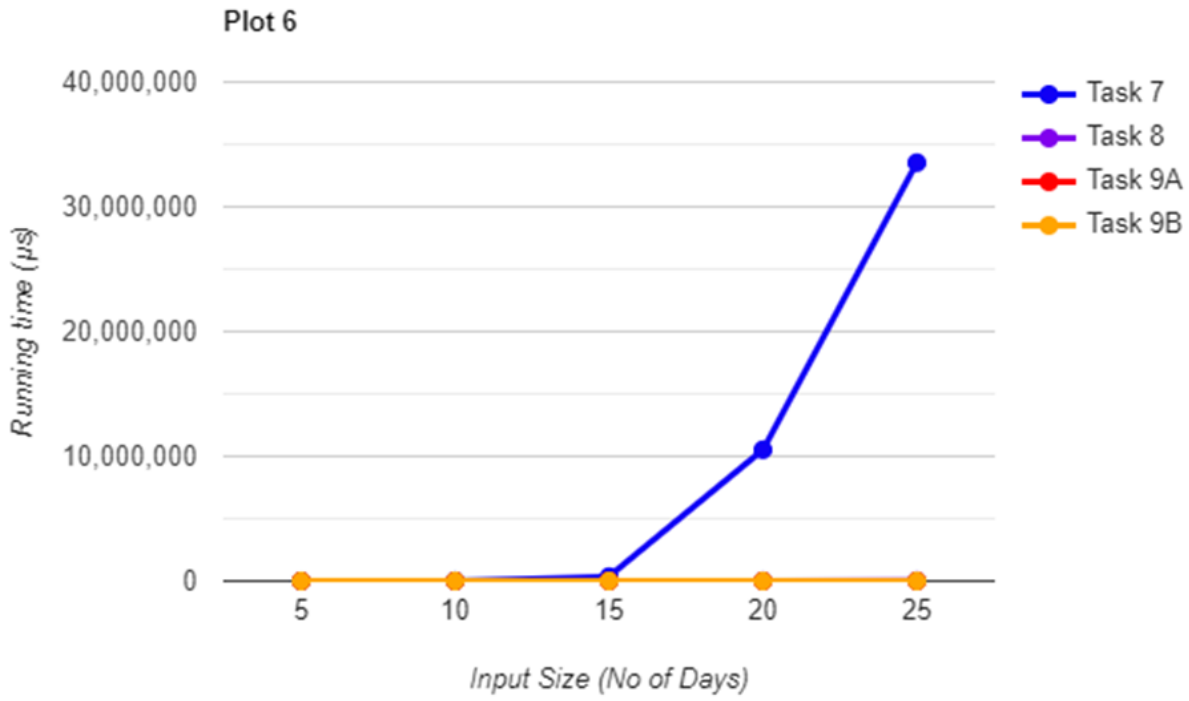


(b)

For both tasks 5, 6a and 6b, the increasing function for running time is in linear time which can be observed in graphs (a) and (b).

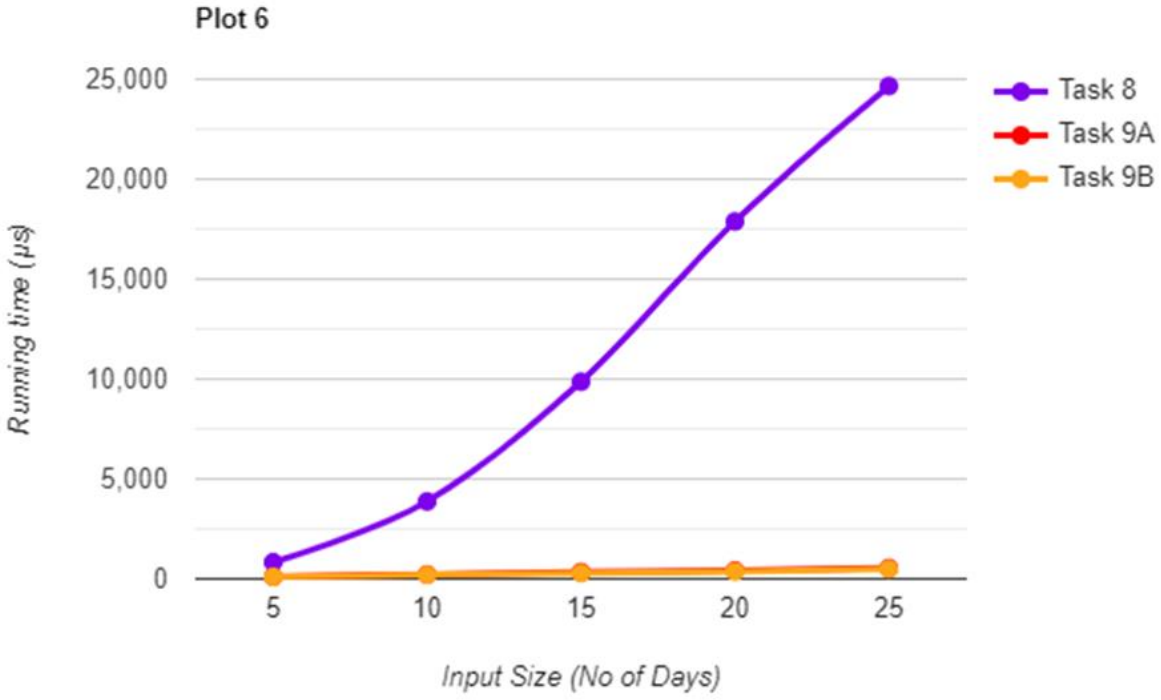
Plot 6

(*Comparison of Task 7, Task 8, Task 9A, Task 9B for variable n, fixed m and fixed c***)**

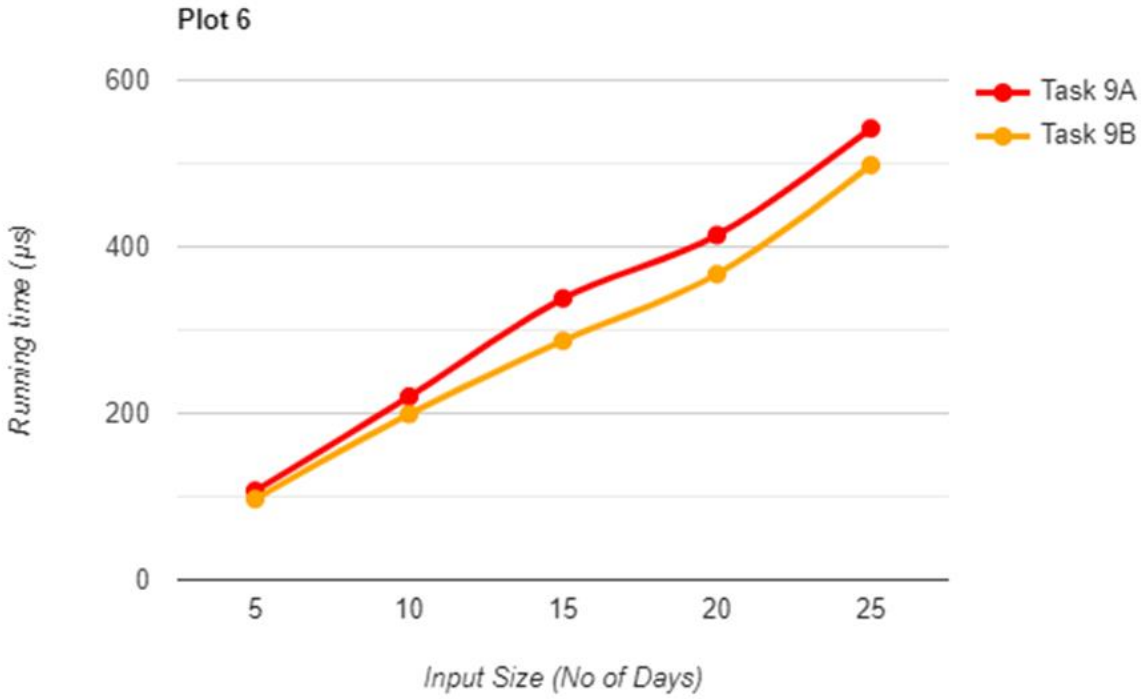


In the Plot 6, we have varied the Number of days(n) and fixed the number of stocks(m) and cooldown period(c)

Using the above graph as reference, we can infer that the Task 7 traces the 2^x graph, as the input size increases the running time increases with power of 2.


Plot 6
25,000
20,000
15,000
10,000
5,000
0
Running time (μs)
5
10
15
20
25
Input Size (No of Days)
Task 8
Task 9A
Task 9B

(a)

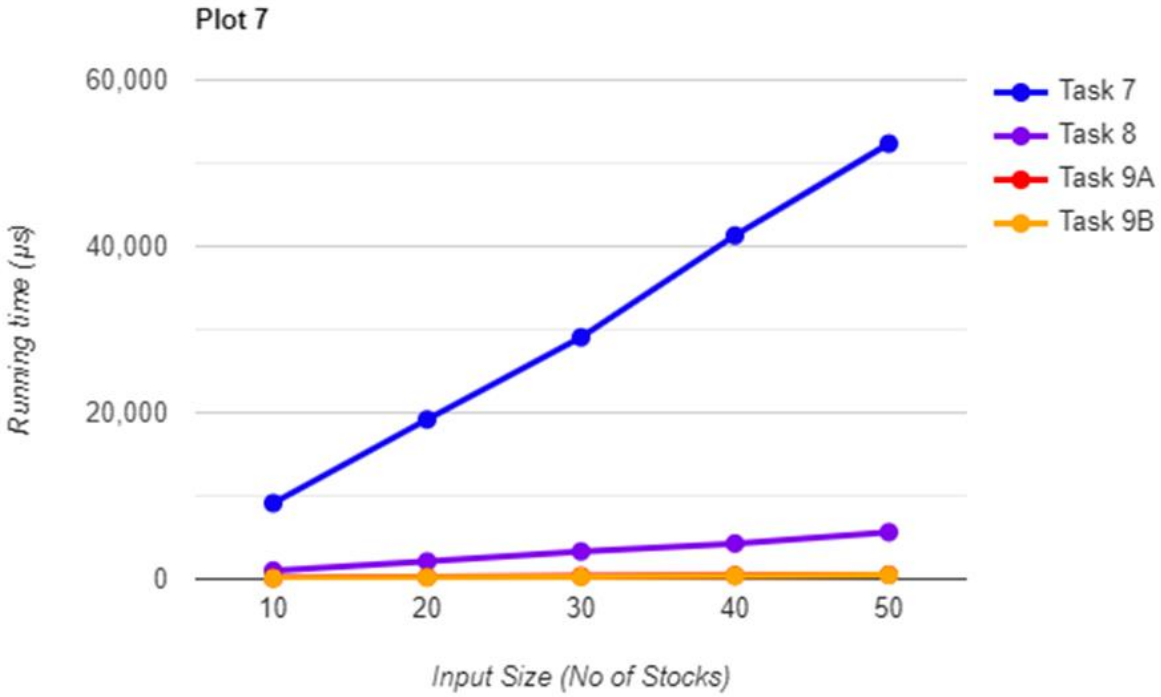


(b)

Referring graphs (a) and (b), we can observe that Task 8 has an increase representing a quadratic function while Task 9A and Task 9B show similar line graphs tracing out linear graph.

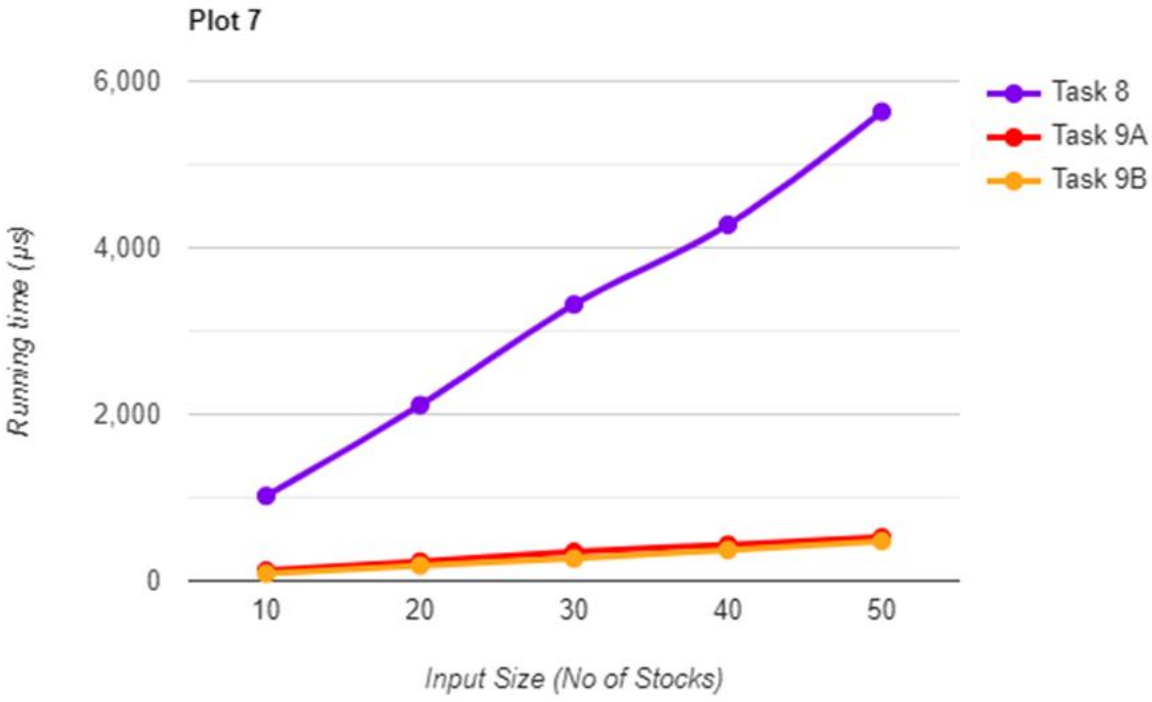
Plot 7

(*Comparison of Task 7, Task 8, Task 9A, Task 9B for variable m, fixed n and fixed c***)**

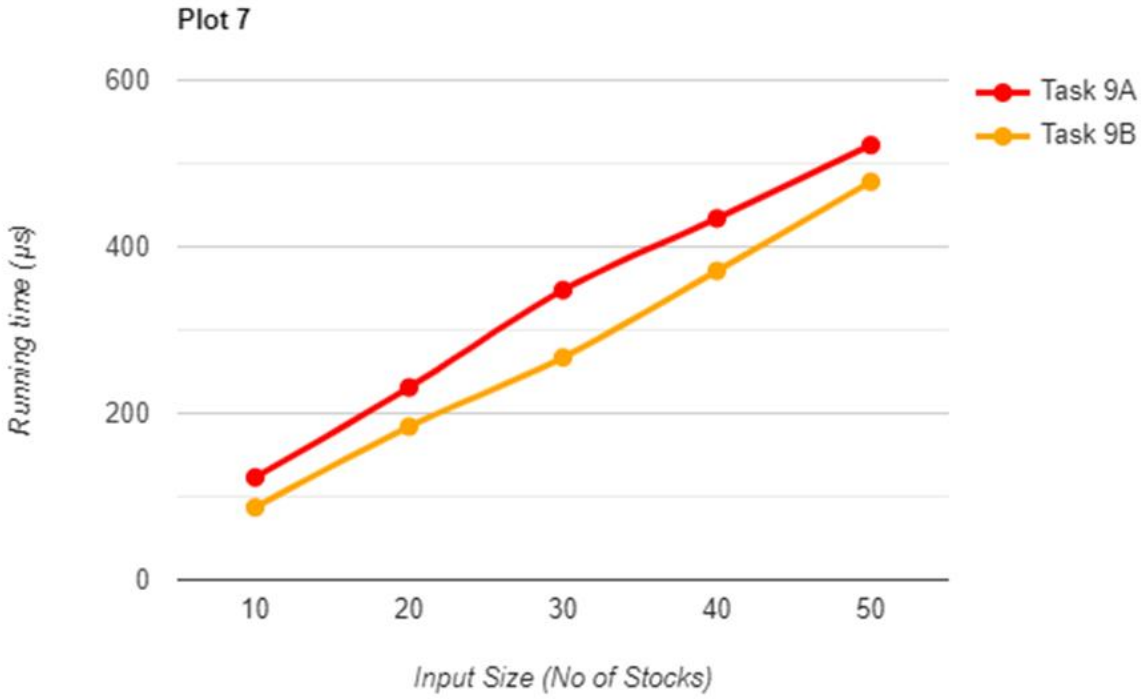


In the Plot 7, we have varied the Number of stocks(m) and fixed the number of days(n) and cooldown period(c)

From the above graph, we can see that Task 7 is increasing linearly with an increase in input Size. This similar trend can be observed for Task 8, Task 9A and Task 9B in below graphs (a) and (b).


Plot 7
6,000
4,000
2,000
0
Running time (μs)
10
20
30
40
50
Input Size (No of Stocks)
Task 8
Task 9A
Task 9B

(a)


Plot 7
600
400
200
0
Running time (µs)
10
20
30
40
50
Input Size (No of Stocks)
Task 9A
Task 9B

(b)

CONCLUSION

LEARNING EXPERIENCE:

The assignment was a great learning experience. We were able to deeply understand how each problem can be solved by a different approach. We were able to learn a greedy approach, recursive approach and dynamic programming using a bottom up and top down approach. How to start with a brute approach considering all the possible combinations and then improving the time complexities by removing redundant repetitions and better algorithms. We were able to observe the time complexities in real time and verify our time complexities. We were able to solve a real world problem as we can apply these algorithms to calculate stock purchases present in the real world as there atre multiple stocks and real people can make limited transactions or make transactions in fwe intervals.

EASE OF IMPLEMENTATION AND PERFORMANCE:

Programming Tasks	Implementation	Performance
Task1	Easy	O(m*n^2)
Task2	Easy	O(m*n)
Task3A	Moderate	O(m*n)
Task3B	Moderate	O(m*n)
Task4	Hard	O(m*n^2k)
Task5	Hard	O(m*n^2*k)
Task6A	Very Hard	O(m*n*k)
Task6B	Hard	O(m*n*k)
Task7	Hard	O(m*2^n)
Task8	Hard	O(m*n^2)
Task9A	Very Hard	O(m*n)
Task9B	Hard	O(m*n)

TECHNICAL DIFFICULTIES:

Programming Tasks	Technical Difficulties
Task1	Did not face much difficulties
Task2	Did not face much difficulties
Task3A	Did not face much difficulties
Task3B	Had some issues coming population the tabulation table and understanding that Kadanes algorithm can be extended here to generate a tabulation data
Task4	Had issues coming up with the recursive tree which will cover all the choice cases and return correct output with termination.
Task5	Had issues choosing the factors to memoize as memoization should reduce the overlapping subproblems and effectively reduce the time complexity
Task6A	Had issues choosing the factors to memoize as memoization should reduce the overlapping subproblems and effectively reduce the time complexity and choose more efficient memoization for better complexity
Task6B	Had some issues with the population of the tabulation table and keeping track of previous values and include the buy sell index
Task7	Had issues coming up with the recursive tree which will cover all the choice cases and return correct output with termination and also include the cooldoen period instead of k transactions

Task8	Had issues choosing the factors to memoize as memoization should reduce the overlapping subproblems and effectively reduce the time complexity and also include the cooldoen period instead of k transactions
Task9A	Had issues choosing the factors to memoize as memoization should reduce the overlapping subproblems and effectively reduce the time complexity and choose more efficient memoization for better complexity and also include the cooldoen period instead of k transactions
Task9B	Had some issues with the population of the tabulation table and keeping track of previous values and include the buy sell index and also include the cooldoen period instead of k transactions