COT5405

Analysis of Algorithms Programming Project Report

Alper Ungor

November21, 2022

Group Members

Rahul Reddy V (UFID 14288319)

Parameswara Reddy A (UFID 40040049)

Sai Krishna Anugu (UFID 42266064)

SECTION 1: TEAM MEMBERS AND CONTRIBUTION:

Team members of this project includes Rahul Reddy Vade , Parameswara Reddy and Sai Krishna Anugu.

Rahul Reddy dealt with the design and implementation of task 1,task 5 and task 6a .The comparative study of task1, task 2, task 3a, task 3b with varying number of days (n) & fixed number of stocks(m) and task1, task 2, task 3a, task 3b with varying number of stocks (m) & fixed number of days(n) was performed by Rahul Reddy and graphs were plotted

Parameswara Reddy contributed with the designing and implementation of task 2, task 3a and task 6b. Parameswara Reddy performed comparative study on task 4, task 5, task 6a, task 6b with varying number of days(n) & fixed values of number of stocks(m),number of transactions(k).

Sai Krishna Anugu handled the analysis & designing , implementation of task 3b, task 4 and task 6a.Sai Krishna Anugu led a comparative study on task 4, task 5, task 6a, task 6b with varying number of stocks (m) & fixed values of number of days(n), transactions(k). Comparative study of task 4, task 5, task 6a, task 6b for varying transactions(k) and fixed number of days and stocks was equally studied by Parameswara Reddy and Sai Krishna Anugu.

We made sure that we divided our work equally such that every team member gets to work on all kinds of tasks. All the team members had an equal contribution for making the Report , Makeup file and source code and zip bundle.

SECTION 2: ALGORITHM DESIGN TASKS

ALG1: Θ(m ∗ n2) time brute force algorithm for solving Problem1

The brute force algorithm for this problem is the nested loop algorithm i.e performing all possible transactions and obtaining the maximum profit.
The outer loop iterates over all the stocks (i.e all rows of the matrix).
The inner loop iterates over a number of days (i.e the columns of the matrix).
For each stock, the algorithm finds the temporary profit by selling on that day. If this temporary profit is greater than the current profit then the current profit is updated with the temporary profit and stock, buy, sell dates with the corresponding values through which maximum profit is achieved. This algorithm returns the stock with maximum profit buy index , sell index of a stock through which maximum profit is achieved.

Algorithm:

```
Initialize (CurrProfit, TempProfit, Stock, Buy, Sell)  to 0
m ← number of stocks
n ← number of days
Prices[ ][ ] ← prices of stocks on various days
For (each stock S starting from 0 to m-1)
        For(each BuyDate starting from 0 to n-2)
                For(each SellDate starting from BuyDate+1 to n-1)
                        If (Prices[S][SellDate] > Prices[S][BuyDate])
                                Update TempProfit ← prices[S][SellDate] -
prices[S][BuyDate];
                        if (TempProfit > CurrProfit)
                                Update CurrProfit ← TempProfit;
                                Update Stock ← S;
                               Update Buy ← BuyDate;
                                Update Sell ← SellDate;
Return(Stock+1, Buy+1, Sell+1);
```

Time complexity: Time complexity of this brute force algorithm is Θ(m*n^2). For the first loop there are m iterations happening, for the second for loop there are n-1 iterations occurring and the third for loop has n-1 iterations.
Since they are nested for loops f(n)= (m)(n-1)(n-1) = mn^2 + m - 2mn.
f(n) is Θ(m*n^2).
Since 0<= c1.g(n) <= f(n) <= c2.g(n).

Space Complexity: To obtain the space complexity of the algorithm , traverse the whole algorithm and check for any memory used.It is clear that extra memory is not used. So space complexity is O(1).

Proof of correctness- Proof by contradiction.

Step1: Assume that the algorithm does not output the correct indice values of stock , buydate, selldate for which maximum profit is obtained.

Step2: The first loop runs across all stocks i.e all rows of the 2D array.

The second inner loop and the inner most loop runs across the first row elements comparing the difference of first element with each and every element of the row and updating the indices of stock, buydate, selldate where maximum difference of two elements is obtained.

But, the algorithm is giving indices for which maximum profit is obtained. This is a contradiction to our assumption that the algorithm does nor give maximum profit.

Step3: Hence, our algorithm gives maximum profit.

ALG-2: Design a Θ(m ∗ n) time greedy algorithm for solving Problem1

The greedy way of solving this problem is updating the minimum stock value available to buy till date for every stock.
Therefore we run the outer for loop on all stocks then initialize maximum profit to zero and assign first day's price to minimum price till date for each stock. Further we run a for loop running on all days for respective stocks. On each iteration, we check for any update on the minimum price (i.e if the present value is less than the set minimum price till date). If so, update minimum price till date to current value and find current profit for the available minimum buy price. Next we compare our current profit to maximum profit and then update the respective value and return the stock, buy, sell indices for which maximum profit is obtained.

Algorithm:

```
Initialize (Ans, Stock, Buy, Sell) = 0
m ← number of stocks
n ← number of days
Prices[ ][ ] ← prices of stocks on various days
For(each stock S starting from 0 to m-1)----------------------------m
        Initialize MaxProfit ← 0
        Initialize MinPriceTillDate ← Prices[s][0]
           For(d starting from 1 to n)-----------------------------------n
                        if (Prices[s][d] less than MinPriceTillDate)
                    Update MinPriceTillDate ← Prices[s][d]
                   Update Buy ← d
                Initialize CurrentProfit = prices[s][d] - MinPriceTillDate
                if (MaxProfit < CurrentProfit)
                      MaxProfit ← CurrentProfit
                      Update Sell ← d
         if (ans less than MaxProfit)
               update   Ans ← MaxProfit
               update Stock ← s
return(Stock+1,Buy+1,Sell+1)
```

Time complexity: Time complexity of this greedy algorithm is Θ(m*n). For the first loop there are (m-1) iterations happening, for the second for loop there are n iterations occurring and the third for loop has n-1 iterations.
Since they are nested for loops f(n)= (m-1)(n) = mn -n.
f(n) is Θ(m*n). Since 0<= c1.g(n) <= f(n) <= c2.g(n).

Space Complexity: To obtain the space complexity of the algorithm , traverse the whole algorithm and check for any memory used.It is clear that extra memory is not used. So space complexity is O(1).

Proof of correctness: Proof by contradiction
Step1: Assume that the algorithm outputs stocks ,buy, sell indices for non- maximum profit.
Step2: Initializes minimum price till date to the first element of the group. The first loop runs over every stock i.e each row of the 2D array. The inner most loop iterates over elements of a row. Here the min price till date variable is compared with the next element in order and updates the minimum price till date to the compared value, if the compared value is less than the minimum price date and buy is updated with this corresponding index .
Step3:Current Profit is obtained by the difference of minimum price till date and the corresponding element of the array. If the current price is larger than the max profit, then the max profit is updated with current price and the sell is updated with corresponding index.
Step4: Max profit of each stock is stored in ans variable and after all iterations are completed the index of stock with maximum profit is returned.
Step5: Here we are getting the index of stock, buy, sell for maximum profit which is a contradiction to our assumption that the algorithm is not outputting the maximum profit.

ALG-3 Design a Θ(m ∗ n) time dynamic programming algorithm for solving Problem1:

For this task we need to write an algorithm using an iterative BottomUp approach. So, we make use of an array to store maximum profit available till date for each stock. I.e the bellman's equation is as follows:
For each stock-> OPT[i] = OPT[i-1] do nothing in day i or buy ticket in day i.
OPT[i] = Math.max(OPT[i-1], prices[i]-smallest), for i > 1
And update smallest price till date if OPT[i - 1] > prices[i] - smallest
We get that maximum profit is the last element of OPT array. After obtaining the maximum profit for every stock, we can compare our final maximum profit and update parallely to get the answer.

Algorithm:

```
Initialize (ans, stock, buy, sell) to  0
m ← number of stocks
n ← number of days
Prices[ ][ ] ←prices of stocks on various days
For (each stock s starting from 0 to m-1)
                Initialize maxProfit←0;
                Initialize opt[n];
                Initialize smallest ← Prices[s][0];
                For (d equal to 1 to n)
                      OPT[d] ←  Math.max(OPT[d - 1], prices[s][d] - smallest);
                      If (OPT[d - 1] greater than prices[s][d] - smallest )
                            Update buy ← d;
                            Update smallest ← Math.min(smallest, prices[s][d]);
                            Update sell ← d;
                 update maxProfit ← OPT[OPT.length - 1];
          If (ans less than maxProfit){
                update ans ← maxProfit;
                update stock ← s;
return (stock+1, buy+1, sell+1);
```

Time complexity of this dynamic algorithm using iterative bottomup is Θ(m*n). For the first loop there are (m-1) iterations utmost, for the second loop there are n iterations occurring.
Since they are nested for loops; f(n)= (m-1)(n) = mn -n.

=>f(n) is Θ(m*n).

Since 0<= c1.g(n) <= f(n) <= c2.g(n).

space complexity: To obtain the **space complexity** of the algorithm , traverse the whole algorithm and check for any memory used. As we made use of an array to store maximum profit till date, we get the space complexity as O(n) because we have n days.

Proof of correctness: proof by induction

Step1: Let us assume the base case as our algorithm is true for m stocks having n days.
Step2: We need to check if our algorithm works for m stocks and n+1 days.
Step3: As we know that our algorithm is true till the nth day, using bellman's equation,for each stock we have the minimum possible stock price to buy till the nth day. We can sell the stock on n+1th day and store the maximum profit possible on n+1th day.
Step4: So this indicates, our algorithm works for m stocks having n days.

Alg4: Design a Θ(m ∗ n 2k) time brute force algorithm for solving Problem2

Brute force algorithm for this algorithm takes O(mn^(2k)). We used two for loops in which the innermost for loop consists of the recursive call that will compare each and every profit that is being obtained by considering two pairs of each stock for every row until utmost k transactions.

Algorithm:

```
initialize m ← number of stocks;
Initialize n ← number of days;
Initialize prices[][] ← prices of stocks
Initialize  brutealg4( a,int i,int buy,int left){
        if(i==n || left==0)
                if(buy==-1 AND left==0)
                        return 0;
                else
                        return -1e9;
        initialize ans ←-1e9;
        if(buy==-1)
                for(each u equal to 0 to m-1)
                        Update ans←max(ans,brutealg4(prices,i+1,u,left)-prices[u][i]);
        else
                for(each u from 0 to m-1)
                        if(u not equal to buy)
                                Update
res←max(ans,brutealg4(prices,i+1,u,left-1)-prices[u][i]+prices[buy][i]);
          Update res←max(ans,brutealg4(prices,i+1,-1,left-1)+prices[buy][i]);
        res←max(ans,brutealg4(prices,i+1,buy,left));
        return ans;
 Initialize maxStockprice( prices,int k)
        return brutealg4(prices,0,-1,k);
Backtrack each recursive call in order to record the indices of stock, buy, sell dates.
The indices with maximum profit are returned.
```

Time complexity: Time complexity of this greedy algorithm is Θ(m*(n^2k)). For the first loop there are (m) iterations happening, for the second loop there are n iterations occurring in which there are n^2k calls happening because of the recursion.
Since they are nested for loops f(n)= m*n^2k
f(n) is Θ(m*n^2k). Since 0<= c1.g(n) <= f(n) <= c2.g(n).

Space Complexity: To obtain the space complexity of the algorithm , traverse the whole algorithm and check for any memory used.It is clear that extra memory is not used. So space complexity is O(1).

Proof of Correctness: Proof by contradiction
Step1: Let us assume that the algorithm gives indices for which maximum profit is obtained. let Maximum profit returned be P.
Step2: Assume that there is some profit M that is greater than P. But in brute force algorithm each pair of stocks in every row is compared and also the profits related to those transactions are compared.
Step3: At some instant of time the profit P and M are compared and P is returned. Meaning P is greater than each and every possible profit and k such profits are returned.

Alg5:Design a Θ(m ∗ n 2 ∗ k) time dynamic programming algorithm for solving Problem2

For this task we need to write an algorithm using an iterative BottomUp approach. The main importance of using dynamic programming is to reduce the time complexity by storing the solutions of the previous computations by storing them in a data structure. The time complexity of this algorithm is O(kmn^2). We are using four for loops for storing values in the 2d dp array. All possible transactions and the respective profit associated with each transaction up to utmost k transactions.

Algorithm:

```
K_Tran ← no of transactions
m ← number of stocks
n ← number of days
Initialize an 2d array Dp_pro of size K_Tran+1,n
For(each r from 0 to K_Tran)
        Set Dp_pro[r][0] to 0
For(each r from 0 to K_Tran)
        Set Dp_pro[r][r] to 0;
For(each t from 1 to K_Tran)
        For(each s from 1 to n)
                Highest_pro ← 0;
                For(each u from 0 to s)
                        for(each r from 0 to price.length)
                                        if((price[r][s] - price[r][u] + Dp_pro[t - 1][u]) >
Highest_pro)
                                        Highest_pro = Dp_pro[t - 1][u] + price[r][s] -
price[r][u]

                                        if(Highest_pro > Dp_pro[t][s - 1])
                                        Dp_pro[t][s] = Highest_pro;
                                        else
                                        Dp_pro[t][s] = Dp_pro[t][s - 1];
r← n-1
t← 0
Set temp = false
while(t<K_Tran and t>=0 and r>0)
        while(Dp_pro[K_Tran][r]==Dp_pro[K_Tran][r-1])
                r--
        Print day on which we sell stock
```

```
Initialize an array ar1 with size m
Initialize an array ar2 with size m

 for(each x from 0 to m)
        Set ar1[x] = Dp_pro[K_Tran][r]-price[x][r];
 for(each y from 0 to m)
        Set ar2[y] = Dp_pro[K_Tran-1][r-1]-price[y][r-1];
 for (each z from p to m)
        if (ar1[z] == ar2[z])
                Set b = r-1
                Print day and stock on which we buy stock
                Set temp = true
                break

 Set u = r-1
 while(temp==false)
        u--
        For(each y from o to m)
                Set ar2[y]=Dp_pro[K_Tran-1][u]-price[y][u];
        For(each z from o to m)
                 if (ar1[z] == ar2[z])
                        Set temp=true;
                        Print day and stock on which we buy

 Increment t
Decrement r
```

Time complexity of this dynamic algorithm using iteration is Θ(k*m*n^2). For the first loop there are (k) iterations utmost, for the second loop there are n iterations occurring and for the third inner loop there are n-2 iterations happening. Similarly for the innermost for loop there are m iterations occuring

Since they are nested for loops;

Thus, the overall complexity is f(n)= k+k+k*n*n*m+k*((m+m+m)+(m+m))

=>f(n) is Θ(k*m*n^2).

Since 0<= c1.g(n) <= f(n) <= c2.g(n).

Space complexity: To obtain the **space complexity** of the algorithm , traverse the whole algorithm and check for any memory used. As we made use of an 2d array to store maximum profit for different transactions, we get the space complexity as O(nk) because we have n days.

Proof of correctness: Proof by induction
Step1: The algorithm gives maximum profit for m number of stocks, n number of days and for k number of transactions.
Step2: Assume, The algorithm gives the maximum profit having different buy and sell days for n transactions
Step3: Now considering n+1 transactions, we get one extra transaction for which our program runs an extra iteration in every 4 loops. And finds the maximum profit without any overlapping.
Step4: Therefore indices for which maximum output is obtained is returned.

Alg6:Design a Θ(m ∗ n ∗ k) time dynamic programming algorithm for solving Problem2

This dynamic programming algorithm using an iterative bottom up approach is more efficient than the alg 5 algorithm. The time complexity of this algorithm is O(m*n*k). In this algorithm we will initialize a 2d dp array which stores maximum profit . where the rows represent the transactions and each column represents maximum profit. The last element of the matrix i.e the last row and last column element is the maximum profit for which the indices need to be returned.

Algorithm:

```
k ← no of transactions
m ← number of stocks
n ← number of days
Initialize array arr1
Initialize an dp array and fill all values using
   dp[i][j] = max(dp[i][j-1],max value of profits for all previous days for every stock)
for( each i from 1 to k)
        For (each j from 1 to k)
                Set max, index each ← -1
                 if(i greater than j) continue;
                for(each k from 0 to m)
                        arr1[k]=max(arr1[],dp[i-1][j-1].first-stockPrices[k][j-1])
                if(stockPrices[l][j]+arr1[l]> greater than maxx)
                Update maximum ←stockPrices[k][j]+arr1[k]
                Update index ← k
                        dp[i][j].first=max(dp[i][j-1].first, maxx)
                if(maxx greater than dp[i][j-1].first)
                                Update dp[i][j].second ←index;
initialize i ← k, j ← n-1;
Backtracking through the dp array
Using while loop till it is true
      if( i or j is 0)
                break
      if(dp[i][j].first is equal to  dp[i][j-1].first)
                Decrement j;
      else
        Set stkNum to dp[i][j].second;
        Add (j,stkNum) to stack
         Set max to dp[i][j].first - stockPrices[stkNum][j];
```

```
        For(each k from j-1 to 0)
              if(dp[i-1][k].first - stockPrices[stkNum][k] is equal to  maxx) {
              Decrement i and assign k to j;
                Add (j,stkNum) to stack and break
  Initialize ele1, ele2;
  while(!stack.empty())
     ele1 ← stack.top();
     stack.pop();
     ele2 ←stack.top();
     stack.pop();
        ans.push_back(temp);
  for(int i=ans.size()-1;i>=0;i--)
    Return  ans[i][0]+1,ans[i][1]+1,ans[i][2]+1;
```

Time complexity of this dynamic algorithm using iteration is Θ(k*m*n). For the first loop there are (k) iterations utmost, for the second loop there are n iterations occurring. Similarly for the innermost for loop there are m iterations occuring
Since they are nested for loops;
Thus, the overall complexity is f(n)= k*m*n+k+n+kn= kmn + kn + k +n>f(n) is Θ(k*m*n).
Since 0<= c1.g(n) <= f(n) <= c2.g(n).

space complexity: To obtain the **space complexity** of the algorithm , traverse the whole algorithm and check for any memory used. As we made use of an 2d array to store maximum profit for different transactions, we get the space complexity as O(nk) because we have n days and k transactions.

Proof of correctness:
Step1: The algorithm gives indices for which the maximum profit is obtained. For the base case of the dynamic programming that is when the transaction k is equal to 0 or the number of days is equal to 1 then the maximum profit that is returned is zero.
Step2: This algorithm considers the profit related to every two pairs of prices and profit is stored. We also initialize that the buydate is less than the sell date and any profit that is greater than the previous profit is updated to current profit and its indices are upaded and this process goes on until the loop terminates.
Step3: Therefore indices for which maximum output is obtained is returned.

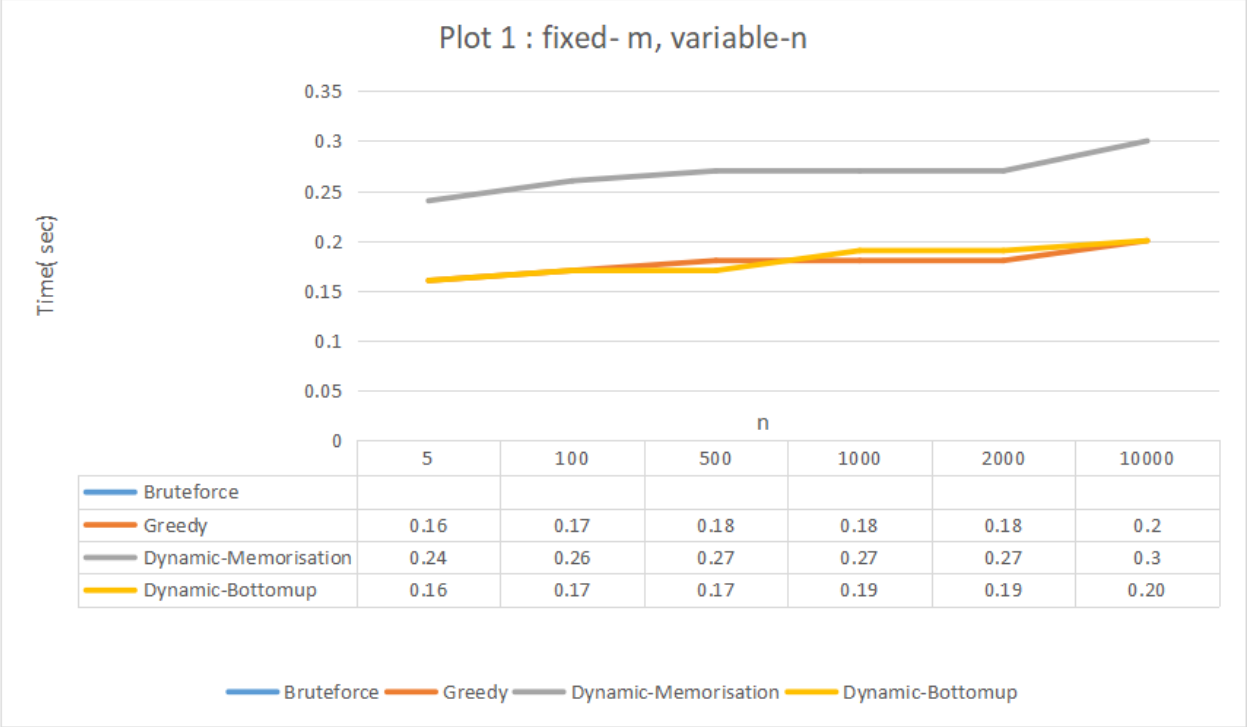
Section 3: Experimental comparative study

For the given two problems we have implemented different algorithms in order to get the most efficient solution with varying time complexities. The brute force algorithm is the most basic approach for implementing the solution with high time complexity . Whereas the greedy algorithm is efficient when compared to that of brute force and also has less time complexity with respect to the brute force algorithm. The dynamic way of approach is the most efficient way of finding the solution. In this project, for problem 1 we are implementing greedy and dynamic algorithms with same time complexities i.e o(m*n). Also in a dynamic way of approach we are using recursive implementation using memoization and iterative implementation using bottom up method. The number of data structures used for each algorithm can differ. The data structures used for brute force algorithm are zero , since we are not storing any values and checking every possible pair for maximum profit. So the space complexity is of constant time O(1). Similarly in greedy algorithm we aren't using any data structures like arrays to store any values. So, the space complexity for greedy algorithm is also of constant time O(1). Whereas in dynamic way of approach we are using array to store values so that we can reduce the time complexity to O(m*n) . But, the space complexity is being increased from O(1) in brute force to O(n) because of the use of array data structure. The two different implementations of the dynamic way of approach for problem 1 are using iterative bottom up(task 3b) and recursive memoization (task 3a). Here both approaches have the same time complexity. The behavior of problem1 algorithms is studied and graphs are plotted for varying and fixed input sizes how the complexities of each algorithm is changing

Similarly for problem2 , the brute force approach has high time complexity and is less efficient . The time complexity of the brute force way of approach to this problem is O(m* n^2k). The space complexity for this approach is O(1). since no data structure is used. The dynamic programming implementation (Task 5) has a time complexity of O(m*n^2 *k) and the space complexity is O(n) because of the use of array to store values. The dynamic programming algorithm with time complexity of O(m*n*k) has two approaches i.e recursive memoization and iterative bottom up. The nature and behavior of these algorithms is studied for varying and fixed input sizes and how the time complexity is being affected for these inputs and graphs are plotted .

Plot1:Comparative study between task1, task 2, task 3A, task 3B

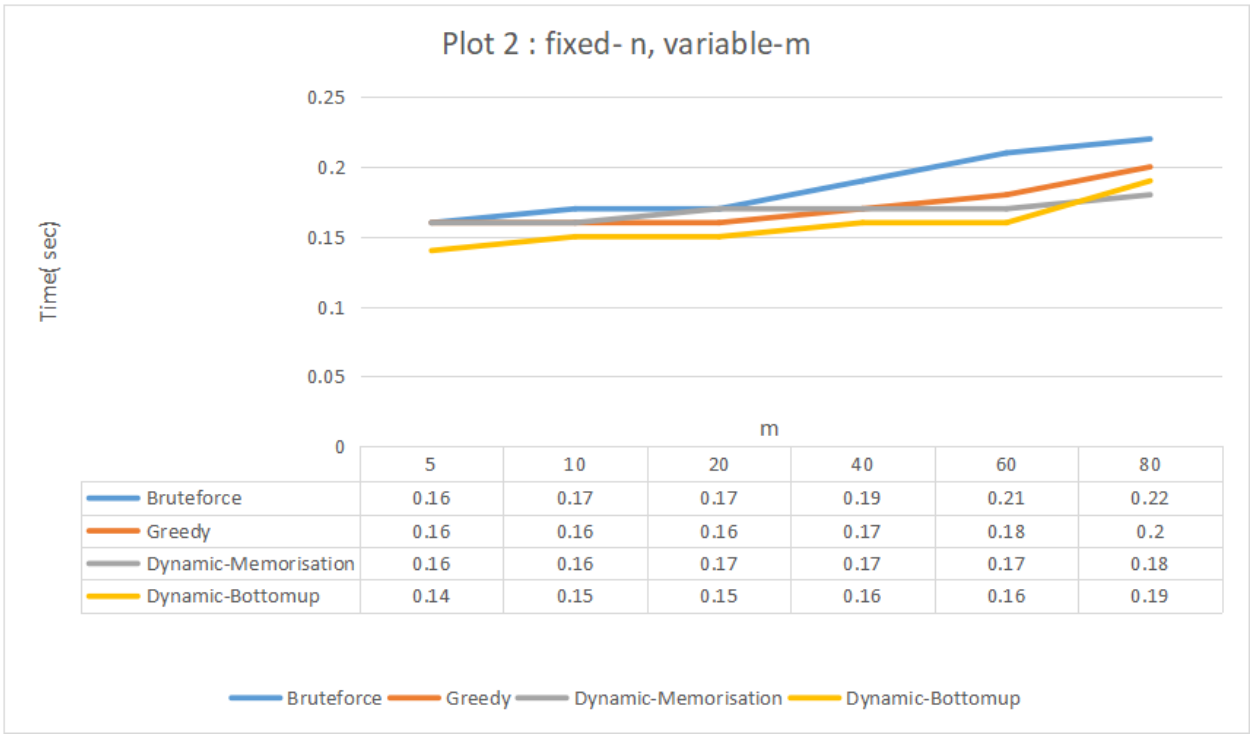
This graph includes the study of comparison between task 1,2,3a,3b. We have made a plot of run time vs number of variables i.e fixed m and variable n. Task1 brute force plot is represented by blue line running on O(mn^2), the red line shows us the variation of time when we use greedy approach having time complexity of O(mn), And when we use dynamic programming by top down and bottom up with time complexity O(mn) we get the plots of grey and orange lines. Thus,we can depict from the graph as brute force way of solving needs the highest time compared to greedy and dynamic approaches. In general, we know that as size increases, the time taken to solve also increases respectively for all approaches. The time increase in brute force increases polynomially, and greedy and dynamic programming increases linearly.



	5	100	500	1000	2000	10000
Bruteforce						
Greedy	0.16	0.17	0.18	0.18	0.18	0.2
Dynamic-Memorisation	0.24	0.26	0.27	0.27	0.27	0.3
Dynamic-Bottomup	0.16	0.17	0.17	0.19	0.19	0.20

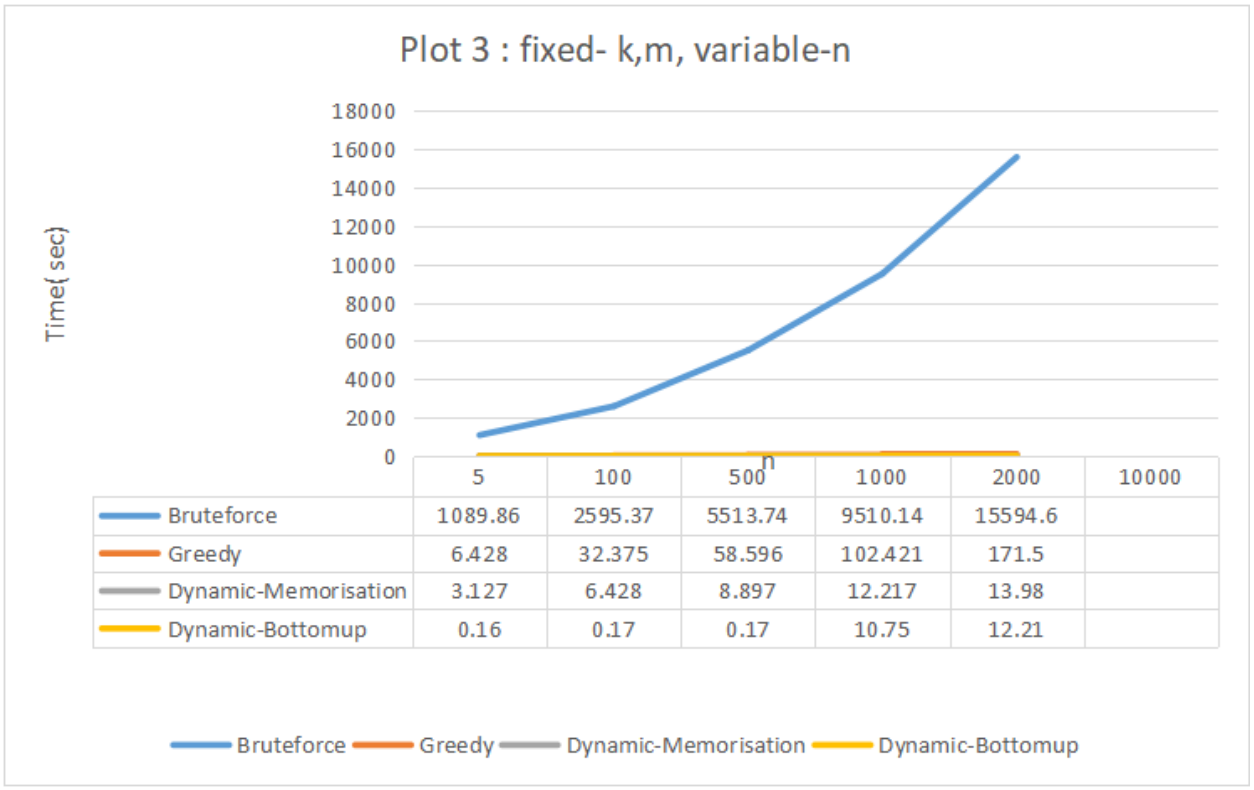
Plot2:Comparative study between task1, task 2, task 3A, task 3B

This graph shows the comparison between tasks 1, 2, 3a, and 3b. We have plotted the run time versus the total number of variables (fixed n and variable m). The task one brute force plot is shown by the blue line, which has an O(mn^2) time complexity. The task two greedy approach plot is shown by the red line, which has an O(mn) time complexity. The task dynamic programming plot, which has an O(mn) time complexity, is shown by the grey and orange lines. As a result, the graph shows that, when compared to greedy and dynamic approaches, brute force methods require the most time to solve problems i.e polynomial. And Greedy and Dynamic programming runs on linear time as the plot rises linearly.



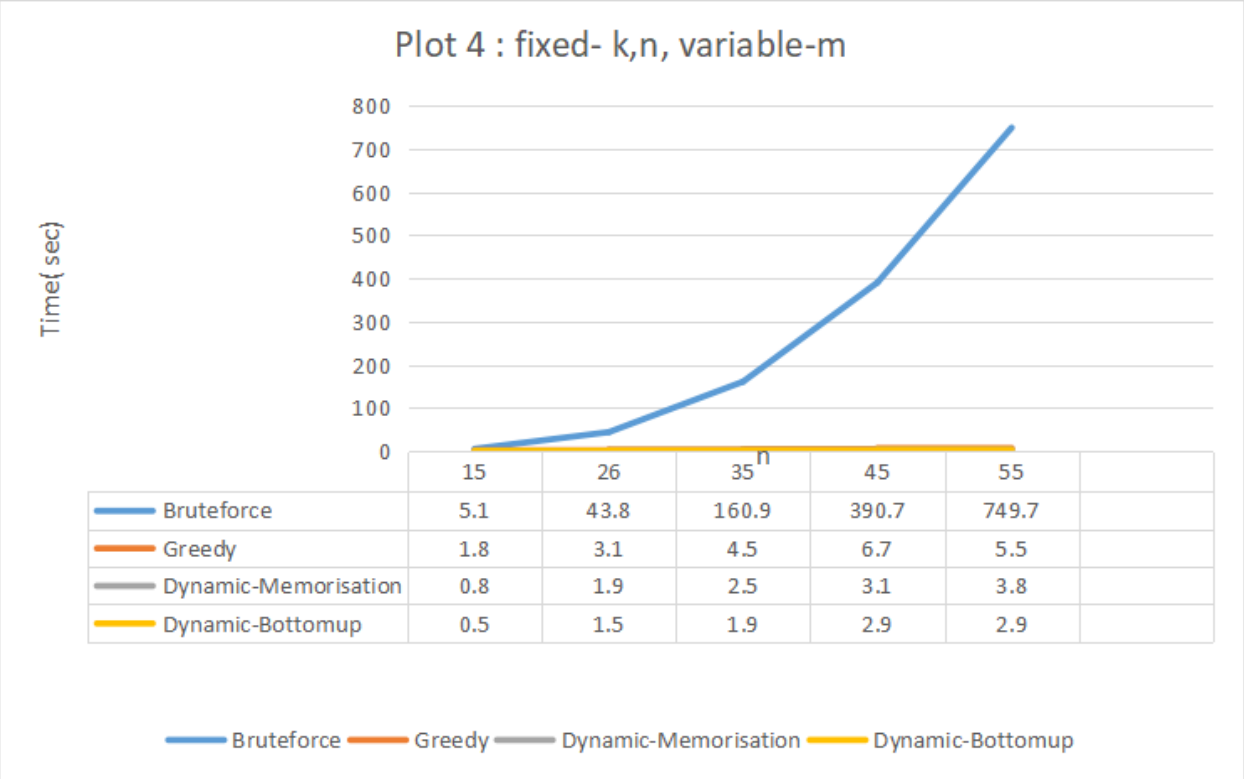
Plot3 Comparison of Task4, Task5, Task6A, Task6B with variable n and fixed m and k

This graph shows the comparison between tasks 4, 5, 6a, and 6b. We have plotted the run time versus the total number of variables (fixed m,k and variable n). The task four brute force plot is shown by the blue line, which has an O(mn^2k) time complexity. The task five dynamic approach plot is shown by the red line, which has an O(k*m*n^2) time complexity. The task six a dynamic programming plot, which has an O(kmn) time complexity, is shown by the grey and orange lines. As a result, the graph shows that, when compared to brute force and dynamic approaches, brute force methods require the most time to solve problems i.e polynomial. And Dynamic programming takes less time.



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

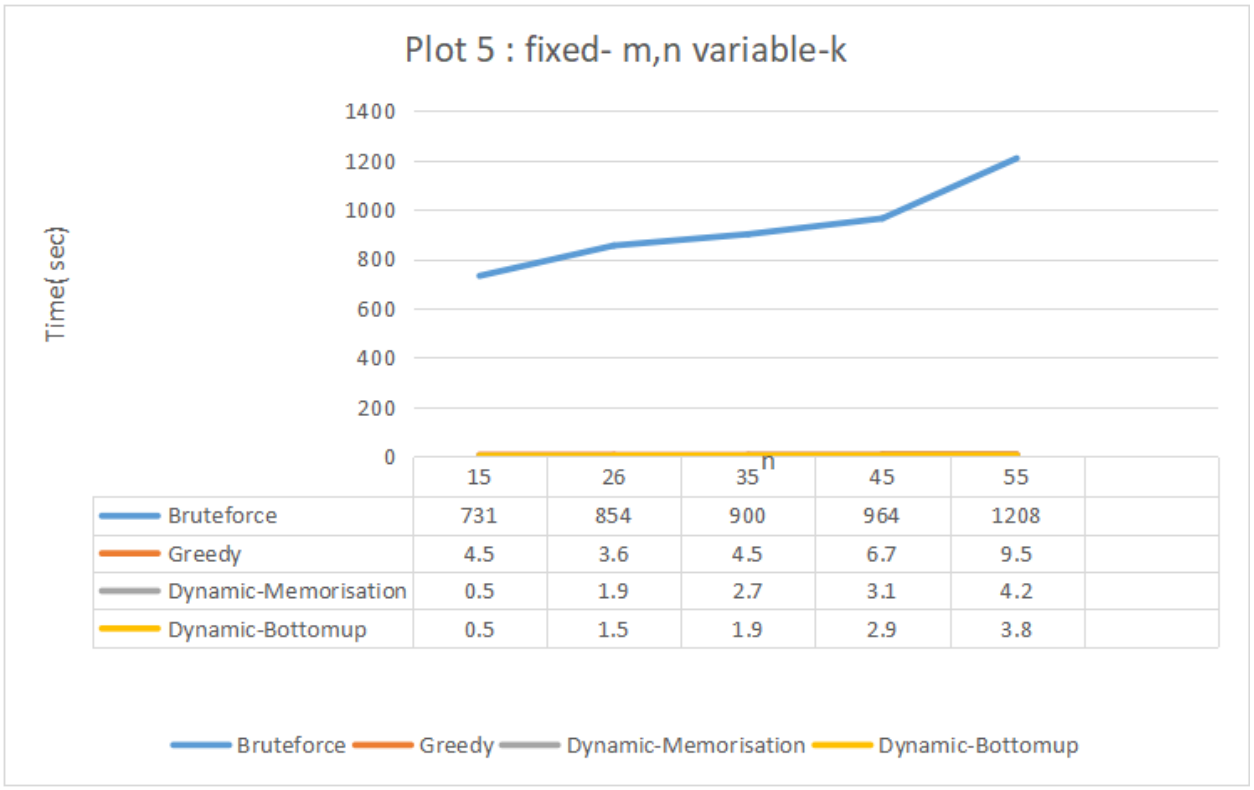
This graph shows the comparison between tasks 4, 5, 6a, and 6b. We have plotted the run time versus the total number of variables (fixed n,k and variable m). The task four brute force plot is shown by the blue line, which has an O(mn^2k) time complexity. The task five dynamic approach plot is shown by the red line, which has an O(k*m*n^2) time complexity. The task six a dynamic programming plot, which has an O(kmn) time complexity, is shown by the grey and orange lines. As a result, the graph shows that, when compared to brute force and dynamic approaches, brute force methods require the most time to solve problems i.e polynomial. And Dynamic programming takes less time.



	15	26	35	45	55
Bruteforce	5.1	43.8	160.9	390.7	749.7
Greedy	1.8	3.1	4.5	6.7	5.5
Dynamic-Memorisation	0.8	1.9	2.5	3.1	3.8
Dynamic-Bottomup	0.5	1.5	1.9	2.9	2.9

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

This graph shows the comparison between tasks 4, 5, 6a, and 6b. We have plotted the run time versus the total number of variables (fixed m,n and variable k). The task four brute force plot is shown by the blue line, which has an O(mn^2k) time complexity. The task five dynamic approach plot is shown by the red line, which has an O(k*m*n^2) time complexity. The task six a dynamic programming plot, which has an O(kmn) time complexity, is shown by the grey and orange lines. As a result, the graph shows that, when compared to brute force and dynamic approaches, brute force methods require the most time to solve problems i.e polynomial. And Dynamic programming takes less time.



SECTION 4 CONCLUSION

Thus, the project assignment we worked on made us implement different approaches for the same problem and observe the usage of time and space complexity on brute force, greedy, dynamic programming both top down and bottom up respectively. As we know the speed and efficiency of solving a problem entirely depends on the algorithm we choose to solve it, this programming assignment helped us to observe different time and space complexities for brute force, greedy, dynamic programming.
A keen observation was made while working on experimental comparative study and we conclude that…in problem 1 and problem2,

Learnings/Challenges TASK 1: Solving problem 1 with a single transaction is a straightforward approach. The challenge faced here is high time complexity, because we need to use 3 for loops then buy and sell all possible combinations to find the maximum profit.
Learnings/Challenges TASK 2 : working on greedy programming to solve problem 1, we made use of a variable to store minimum stock value to buy for each stock. Compared to task1, we have a better time complexity of .
Learnings/Challenges TASK 3A : By using an array to store optimal profit for each day, we can find the maximum profit for each stock and update the same for all stocks. The challenge here is as we are working on recursion we couldn't make our code work for more than 10,000 days. This is due to a recursive call function being called for more than 10,000 times, we are facing a stack overflow issue. If we observe the greedy (task 2) and task 3a, 3b it is clear that greedy is giving optimal and efficient solutions. So, there is no need to use dynamic programming because the time complexity is same.
Learnings/Challenges TASK 3B : Using iterative BottomUp, the challenge faced is constructing the bellman's equation. To overcome this, we solved each stock and found the maximum profit for each stock and update the final maximum profit while working on each stock
Learnings/Challenges TASK 4 : Brute force algorithm for problem2, where k transactions are involved is not an easy implementation. Because it needs to check every possible pair of prices for a particular stock and needs to return the utmost k maximum transactions that give the maximum profit. Adding the recursive part in the for loop was the difficulty which we faced. Returning the maximum profit is easy but backtracking the indices for which maximum profit is obtained is hard because of the recursive calls happening in the nested for loops.
Learnings/Challenges TASK 5 : Using an iterative bottom up approach for a dynamic programming algorithm which runs in O(kmn^2) . The challenge we faced here is while doing dynamic programming, we were to use four for loops. The backtracking to get the indices i.e sell and buy dates for the maximum profit is a hard part to solve in this task.

Learnings/Challenges TASK 6A: This dynamic programming algorithm using recursive memoization takes O(k*m*n) . In this algorithm we are using a 2d dp array to store the maximum profits for k transactions and for every possible pair of prices in each row. So a lot of learning was required.

Learnings/Challenges TASK 6B:We used a three dimensional array, where each element gives us the profit for a particular pair of buy and sell for all stocks. The challenge faced here is performing the backtracking to get the indices i.e sell and buy dates for the maximum profit.