COT 5405 - Analysis of Algorithms

Programming Project

Group Members
Veera Rajasekhar Reddy Gopu- UFID: 52826538
Hrushyang Adloori- UFID: 86776220
Goutham Kumar Mekala- UFID: 29386297

Fall Semester 2023
21st November 2023

Team members:

1.Veera Rajasekhar Reddy Gopu (UFID:52826538)

Worked on algorithm development, mathematical formulation, analysis, documentation, writing code and generating graphs.

2. Hrushyang Adloori (UFID:86776220)

Worked on algorithm development, mathematical formulation, analysis, documentation, writing code and generating graphs.

3. Goutham Kumar Mekala (UFID: 29386297)

Worked on algorithm development, mathematical formulation, analysis, documentation, writing code and generating graphs.

Design and Analysis of Algorithms:

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

Problem2 Given a matrix A of m × n integers (non-negative) representing the predicted prices of m stocks for n days and an integer k (positive), find a sequence of at most k transactions that gives maximum profit.
[Hint :- Try to solve for k = 2 first and then expand that solution.]

Alg1: Θ(m*n^2) complexity Brute force algorithm for solving Problem1

Design:

1. The brute force approach is quite straight forward for the Problem1. We iterate through every possible combination of a single valid transaction.
2. Valid transactions can be defined as when buyDate <= sellDate for a particular stock.
3. The outermost loop iterates through every stock assigning a particular stockId.
4. After the stockId is selected, we find the maximum profit obtained from that stock by iterating over each day's price of that stock.
5. The second loop iterates over all days of that stock. To obtain maximum profit, we run another loop for the sellDate where it iterates from the buyDate to the rest of days i.e. until the last day.
6. In the innermost loop i.e. iterating through all the prices from the buyDate to remaining days, we keep track of local maximum by taking the difference in stock prices from the obtained buyDate and sellDate.

7. After finding the maximum profit for each stock i.e. comparing the local maximums and taking maximum at each level. We compare it with the global maximum. Hence producing the maximum profit for a single transaction while recording it and considering all stocks

Pseudo-Code:

```
Task1(m,n,priceList)
    For stockIndex = 0 to m
    For buyDate = 0 to n
    For sellDate = buyDate+1 to n
    if(maxProfit < priceList[stockIndex][sellDate] - priceList[stockIndex][buyDate])
        Update -> SellDate, BuyDate, StockIndex.
Return transactionParameters.
```

Time and Space Complexity:

The time complexity of this algorithm would be θ(m*n^2) as we are iterating through three nested loops with worst case iterations going up to m, n and n-1.
The space complexity for this algorithm is O(1) as we are using only a constant number of variables independent of m & n.

Proof of Correctness:

Here, the algorithm keeps track of all possible valid transactions. Consider this a set.
Let's assume there is a valid transaction not part of this set that gives maximum profit. But our initial set consists of all possible valid transactions. Hence making it invalid transaction, which is a contradiction to our initial assumption. Therefore, by proof by contradiction, our algorithm is bound to produce this transaction. Hence, this algorithm to be consistent and correct.

Alg2: Θ(m*n) Complexity Greedy algorithm for solving Problem 1

Design:

- For Algorithm 2, we wrote a greedy algorithm. In this, we find the maximum profit.
- Initialize parameters such as maxProfit and minimumPrice and set these to -1 to find out the maximum profit that can be gained at the beginning.
- Also, initialize transactionStockId, transactionBuyDate, transactionSellDate to keep track of the buying and selling dates of the stock to achieve maximum profit.
- The outer loop iterates through every stock assigning a particular stockId.
- Start minimum price as the starting day price of the current stock.
- The inner loop iterates through every day starting from day 1 to the last day.
- Here, Keep track of the minimum stock price, so that we can track the buy date. If the price of the stock on a particular day is less than the minimum price, update the minimum price and the buy date.
- Also, calculate the current profit by subtracting the price of the stock at a particular day with the minimum profit.
- If current profit is greater than the maximum profit, update the maxProfit with current profit and also keep updating the buyDate and sellDate variables.
- After iterating through both the loops, we can get the Maximum profit that can be achieved and the buying and selling date of the stock to achieve this profit.

PseudoCode:

```
Task2(m,n,priceList)
    For stockIndex = 0 to m
        minimumPrice = priceList[stockIndex][0]
        For sellDate = 1 to n
            if(currentPrice < minimumPrice)
                Update -> minimumPrice
```

if(maxProfit < currentProfit)

Update -> maxProfit

Update -> SellDate, BuyDate, StockIndex.

Return transactionParameters.

Time and Space Complexity:

- Here we keep track of minimum price because we get the maximum profit only when we buy the stock on the day with minimum price. So, we keep track of that minPrice variable for each stock, Which proves that the time complexity is Θ(mn).
- We are just storing the minimum price and maximum profit, and as we need only one such transaction, the space complexity is O(1).

Proof of Correctness:

We are tracking local minimum at each day and fixing it as buy date. And from there searching for global maximum at that particular selected stock. Since we have the choice to make only one transaction, this works consistently as we are simultaneously checking for max profit for each selected transaction and stock as well. Hence yielding us the optimal global max profit.

Alg3: Θ(m*n) complexity Dynamic Programming algorithm for solving Problem1

Design:

- Initialize the transaction parameters for StockId, BuyDate, SellDate, minimumPrice and maxProfit with least integer value possible.
- Declare a 2-Dimensional table with mxn where m is the number of stocks and n is the number of days.
- Our aim is to fill this 2-D dp table so that we can later backtrack for the exact transaction to be selected.
- We iterate over each stock in the outer loop, we again take a stockMemoization list for each iteration.
- This represents the maximum profit across a particular stock until i th day in stockMemoization[i].
- To do this we iterate over all the days in the inner loop and fill the table by taking the maximum of 0 or stockMemoization[i-1] + priceList[stockid][i] - priceList[stockid][i-1].
- At the end of iteration of one stock the maximum profit obtained from that stock is assigned to 2D dp at that index corresponding to that stock.
- For extracting the transactions from the DP table, we make use of backtracking logic.
- For backtracking, we initiate globalMax with minimum value. Iterate over the whole mXn table using two nested loops iterating over m and n respectively.
- If we observe a change in the value being greater than globalMax we record the transaction.

Mathematical Formulation:

$M[i, j]$

$case\ 1\colon\ 0 \qquad if\ j\ =\ 0\ \ or\ \ i\ =\ 0$

$case\ 2\colon\ max(0,\ M[i, j\ -\ 1]\ +\ p[i, j]\ -\ p[i, j\ -\ 1] \qquad otherwise$

$Goal\colon\ max(M[i, j])\ ,\ where\ 1 \leq i \leq m\ and\ 1 \leq j \leq n$

Pseudo Code:

```
Task3b(m,n,priceList)
    For i = 1 to m
        M[i][0] <- 0
        For j = 1 to n
            M[i][j] <- max(0, M[i][j-1] + pij - pij-1)

Task3a(m,n,priceList)
M[m][n] <- -1
 For i = 1 to m
    M[i] = mCompute(n)
    mCompute(j)
    if(j==0)
    Return 0
    if(M[i][j] is uninitialized)
        M[i][j] <- max(0, mCompute(j-1) + pij - pij-1)
    Return M[i][j]
```

BackTracking:

```
FindSolution(m,n,priceList,M)
    MaxProfit = -inf
```

For i = 0 to m

For j =0 to m

If (maxProfit < M[i][j])

stockIndex<- i, sellDate<-j

buyPrice = price[stockIndex][sellDate] - maxProfit

buyDate = Index(buyPrice)@priceList

Time and Space Complexity:

- The time complexity of this algorithm is θ(m*n) as we only fill the DP table of size mXn using two nested loops. We iterate to a maximum of m and n respectively.
- The space complexity of this algorithm is also O(m*n) as we are at max a 2D list for filling the DP table.

Proof of Correctness:

Here we consider our subproblem to be memoization[i][j] which can be defined as maximum profit achieved until j th day and considering stocks until i th stock. The base case for memoization[0][0] is 0 vacuously(i.e. If either i or j = 0). Basically, as we proceed taking a single step down in the recursion tree, at least one of i or j is decrementing by 1. Regardless of whatsoever child of subproblem, we eventually hit the base case. This means that if we use those subproblems as our base cases, we can make sure that every branch of the induction tree ends at a base case where all of the assumptions are met. We fill the table based on previous optimal values. Hence by definition of the subproblem memoization[m][n] should produce the maximum profit for the whole. So by the property of optimal substructure, our solution is bound to be correct.

Alg4: Θ(m*n^(2*k)) complexity Brute Force Algorithm for solving Problem2

Design:

- We followed a recursive approach to exhaust all possible combinations of k transactions across all stocks.
- The base case for the recursive function is returning 0 if either n (n th day) or k (k th transaction).
- Then for any other case a recursive call is made with parameters n-1 and k and the returned value is stored in maxVal.
- Then we iterate over the number of stocks in the outer loop, followed by a nested loop which iterates over from n-1 to 0.
- In the nested loop, the maxVal is updated with the maximum of previous maxVal & return value of recursive call made with parameters the inner loop iterating variable (i.e. j that iterates from n-1 to 0) and k-1 (indicating completion of a transaction).
- Here we are filling a mXn table recursively with the corresponding parameter calls and taking maximum at each level. Hence the table can be backtracked for the solution.

Pseudo Code:

```
Task4(m,n,k,M)
    If(n==0 || k==0)
        M[k][n] <- 0
    prevDayMax = Task4(m,n-1,k,M)
    For i = 1 to m
        For j = n-1 to 1
            prevDayMax = max(prevDayMax, Task4(m,j,k-1,M)+ pin - pij)
    Return M[k][n] = prevDayMax
```

Time and Space Complexity:

- The time complexity for this brute algorithm is $\Theta(m*n^{(2*k)})$ as we are generating recursion tree of height n^k and work done at each level is m*n^k. Hence giving us the complexity stated above.
- The space complexity of this algorithm is O(n*k) as every recursion call needs memory in the stack.

Proof of Correctness:

Brute force algorithm we implemented finds the maximum profit by exhaustion. All possible cases where we can make k number of transactions which are non-overlapping across all stocks are checked. Hence consistently giving the maximum profit along with the transactions.

Alg5: Θ(m*n^2*k) complexity with Dynamic Programming algorithm for solving Problem2

Design :

- In this algorithm we construct the sub-problem to be M[i][j] which is defined as the maximum profit achieved until i th day by performing at most j transactions considering transactions across all stocks.
- Our goal is to compute M[n-1][k] i.e. maximum profit achieved until the last day by performing at most k transactions across all stocks.

  GOAL => Compute M[n-1][k]
- Base cases can be described as when either i or j is equal to zero then M[i][j] is assigned zero.
- Otherwise, the value for M[i][j] is maximum of previously computed value i.e. M[i-1][j] & maximum profit achieved across all stocks until i-1 th day (stock price of a th stock at i th day - stock price of a th stock at b th day where a & b are iterated through 1 to m and 1 to i-1 respectively) added to M[b][j-1]. (Refer the Mathematical Formulation)
- After achieving all the values for M[n-1][k], the corresponding values can be stored in the Memoization table. This table can be used to back track the transactions responsible for the profit achieved at that position.
- This can be done using the BackTracking logic described in the pseudo code. Hence obtaining the exact transactions that return maximum profit.

Mathematical Formulation:

$M[i, j]$

$Case\ 1\colon\ 0 \qquad if\ i = 0\ \ or\ j = 0$

$Case\ 2\colon\ max(M[i - 1, j],\ M[b, j - 1]\ + p[a, j]\ - p[a, b] \qquad otherwise$

$where\ n - 1 >=\ b\ >=\ 1,\ 1 <= a <= m$

$Goal\colon\ Find\ M[n - 1, k]\ which\ is\ the\ maximum\ profit$

$where\ n\ =\ total\ no\ of\ days\ \ and\ \ k\ =\ number\ of\ transactions$

Pseudo Code:

```
Task5(m , n , k, p)
    For i = 1 to k
        M[i][0] = 0
    For i = 1 to n
        M[0][i] = 0

    For i = 1 to k
        For j=1 to n
            For  k = 1 to m
                For j1 = n-1 down to 0
                    M[i][j] <- max(M[i][j],  M[i-1][j] + pkj - pkj1)
```

Time and Space Complexity:

- The time complexity of this algorithm is Θ(m*n^2*k) as we are filling the DP table of dimension nXk and to fill each element the work done is m*n.
- The space complexity of this algorithm is O(n*k) as we need to store the 2D dp table.

Proof of Correctness:

Here we consider our subproblem to be M[i][j] which can be defined as maximum profit achieved until i th day and performing at most j transactions considering all the stock. The base case for [0][0] is 0 vacuously(i.e. If either i or j = 0). Basically, as we proceed taking a single step down in the recursion tree, at least one of i or j is decrementing by 1. Regardless of any child of subproblems, we eventually hit the base case. This means that if we use those subproblems as our base cases, we can make sure that every branch of the induction tree ends at a base case where all of the assumptions are met. We fill the table

based on previous optimal values. Hence, by definition, the subproblem M[m][n] should produce the maximum profit for the whole. So by property of optimal substructure, our solution is bound to be correct.

Alg6: Θ(m*n*k) complexity with Dynamic Programming algorithm for solving Problem2

Design :

- In this algorithm we construct the sub-problem to be M[i][j] which is defined as the maximum profit achieved until i th day by performing at most j transactions considering transactions across all stocks.
- Our goal is to compute M[n-1][k] i.e. maximum profit achieved until the last day by performing at most k transactions across all stocks.

  GOAL => Compute M[n-1][k]
- Base cases can be described as when either i or j is equal to zero then M[i][j] is assigned zero.
- Otherwise, the value for M[i][j] is maximum of previously computed value i.e. M[i-1,j] & maximum profit achieved across all stocks until i-1 th day (stock price of a th stock at i th day - stock price of a th stock at b th day where a & b are iterated through 1 to m and 1 to i-1 respectively) added to M[b][j-1].
- Here to reduce the repetitive checking, we introduce a new variable from the start i.e maxDiff. Here if we already take the maximum of differences between M[a][j-1] and stock price[m][a] at every i th day iteration. We can eliminate the nested loop that iterates through 1 to i-1. Hence scaling down the time complexity from n^2 to just n.(Refer the Mathematical Formulation)
- After achieving all the values for M[n-1][k], the corresponding values can be stored in the Memoization table. This table can be used to back track the transactions responsible for the profit achieved at that position.

- This can be done using the BackTracking logic described in the pseudo code. Hence obtaining the exact transactions that return maximum profit.

Mathematical Formulation:

$M[i, j]$

$case\ 1\colon\ 0 \qquad if\ i = 0\ or\ j = 0$

$case\ 2\colon\ max(M[i - 1, j],\ MaxDiff[s]\ +\ p[s, j]) \qquad otherwise$

$MaxDiff[s]\ =\ max(MaxDiff[s],\ M[i, j - 1] - p[s, j])$

$where\ 1\ <=\ s\ <=\ m$

Pseudo Code:

```
Task6a(m,n,k,p)
    M[k+1][n] <- -1
    D[k+1][m] <- 0
    For i = 0 to k
        For j= 0 to m
            D[i][j] = -p[i][0]
    Return mCompute(n-1, k,D,M)

mCompute(n,k,D,M)
    if(n==0 || k==0)
        Return 0
    if(M[k][n] is uninitialized)
        maxVal = mCompute(n-1, k, M, D)
        For i = 0 to m
            D[k][i] = max(D[k][i], mCompute(n,k-1,D,M) - p[i][n])
            maxVal = max(maxVal, D[k][i] + p[i][n])
```

```
            Return M[k][n] = maxVal
        Return M[k][n]
Task6b(m ,n, k, p)
M[k+1][n] <- -1
M[0][n] = 0
M[k][0] = 0

For t= 1 to k+1
        For i = 0 to m
        D[i] = -p[i][0]
For d = 1 to n
        maxVal = M[t][d-1]
        For s = 1 to m
                maxVal = max(maxVal, p[s][d] + D[s])
                D[s] = max(D[s], M[t-1][d] = p[s][d])
        M[t][d] = maxVal

BackTracking(M, Sol)
t = k, d = n-1
while(True)
        if(t==0 || d==0)
                break
        if(M[t][day] == M[t][day-1]
                day = day-1
        else
                For s = 0 to m
                        For j = d-1 down to 0
                                if(M[t-1][j] + p[s][d] - p[s][j] == M[t][d])
                                        Sol<- {s,j,d}
```

t = t-1

d = j

break

Return Sol.

Time and Space Complexity:

- The time complexity of this algorithm is Θ(m*n*k).We need to fill the 2D dp table of size nXk but, as we are eliminating the nested inner loop which iterates to n at worst case.The work done to fill each element is only m.
- The space complexity of this algorithm is O(m*n*k).

Proof of Correctness:

Here we consider our subproblem to be M[i][j] which can be defined as maximum profit achieved until i th day and performing at most j transactions considering all the stock. The base case for [0][0] is 0 vacuously (i.e. If either i or j = 0). Basically, as we proceed taking a single step down in the recursion tree, at least one of i or j is decrementing by 1. Regardless of considering any child of subproblems, we eventually hit the base case. This means that if we use those subproblems as our base cases, we can make sure that every branch of the induction tree ends at a base case where all of the assumptions are met. We fill the table based on both previous optimal values and optimally calculating at that level. Hence by definition of the subproblem M[m][n] should produce the maximum profit for the whole. So by property of optimal substructure, our solution is bound to be correct.

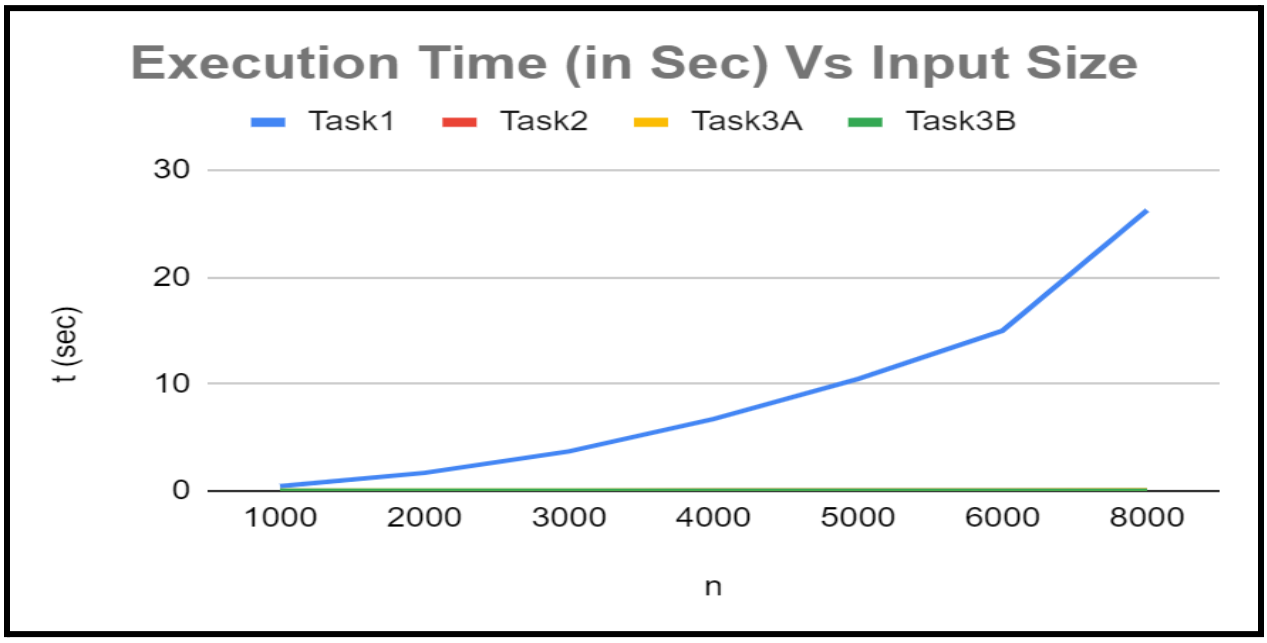
Experimental Comparative Study:

We analyzed the algorithm's execution time and made comparison graphs using various input sizes and randomly generated inputs. The performance tables and graphs are shown below.

Table 1: Execution Time (in Sec) Vs Input Size for Tasks 1 - 3b taking **m=70**

n	Task1	Task2	Task3A	Task3B
1000	0.435937	0.000913	0.003567	0.003124
2000	1.691829	0.001708	0.008045	0.005495
3000	3.702986	0.002605	0.012578	0.007749
4000	6.731998	0.00358	0.015136	0.011247
5000	10.483839	0.004373	0.019123	0.014229
6000	15.005923	0.005146	0.021815	0.017792
8000	26.273349	0.006912	0.031006	0.021076

Graph 1.1: Execution Time (in Sec) Vs Input Size for Tasks 1 - 3b taking **m=70**



Because the execution time values in the preceding graph are overshadowing, we generated another graph to compare Task 2, Task 3a, and Task 3b.

Graph 1.2: Execution Time (in Sec) Vs Input Size for Tasks 2- 3b taking **m=70**

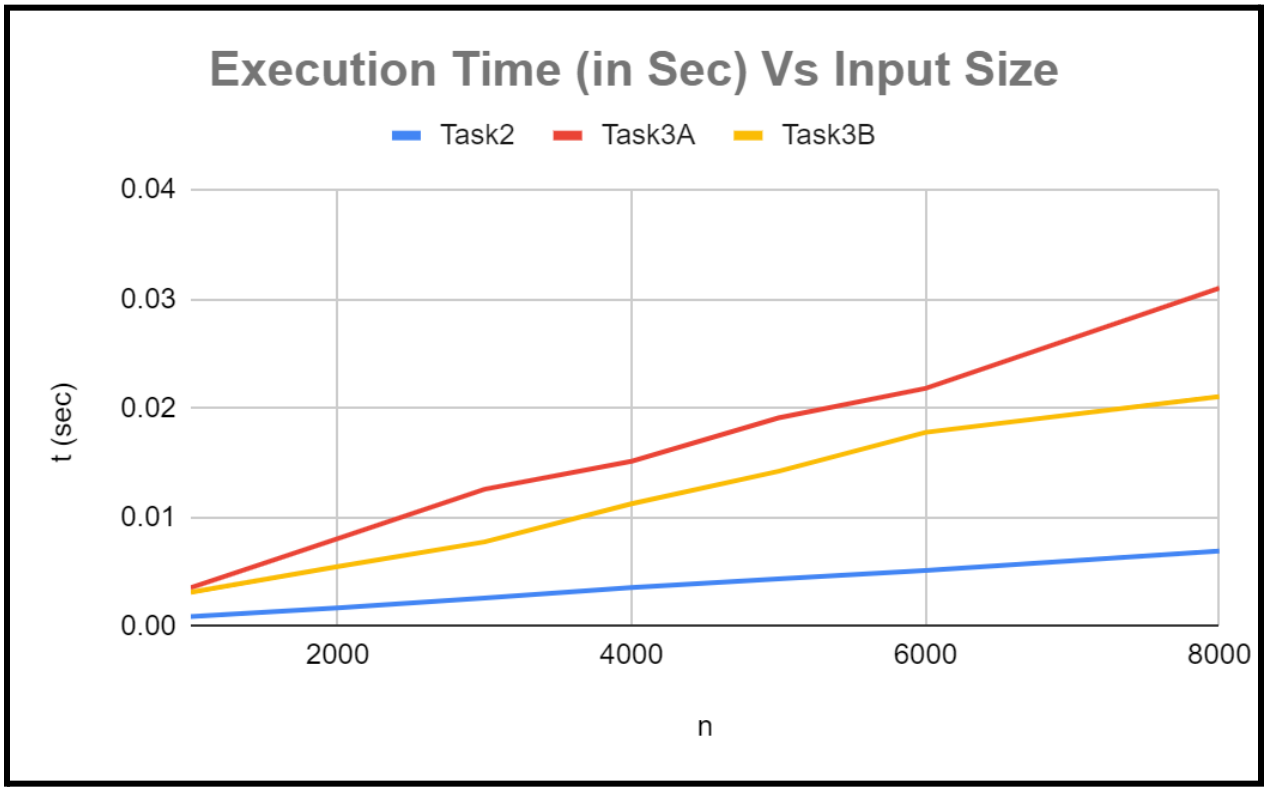
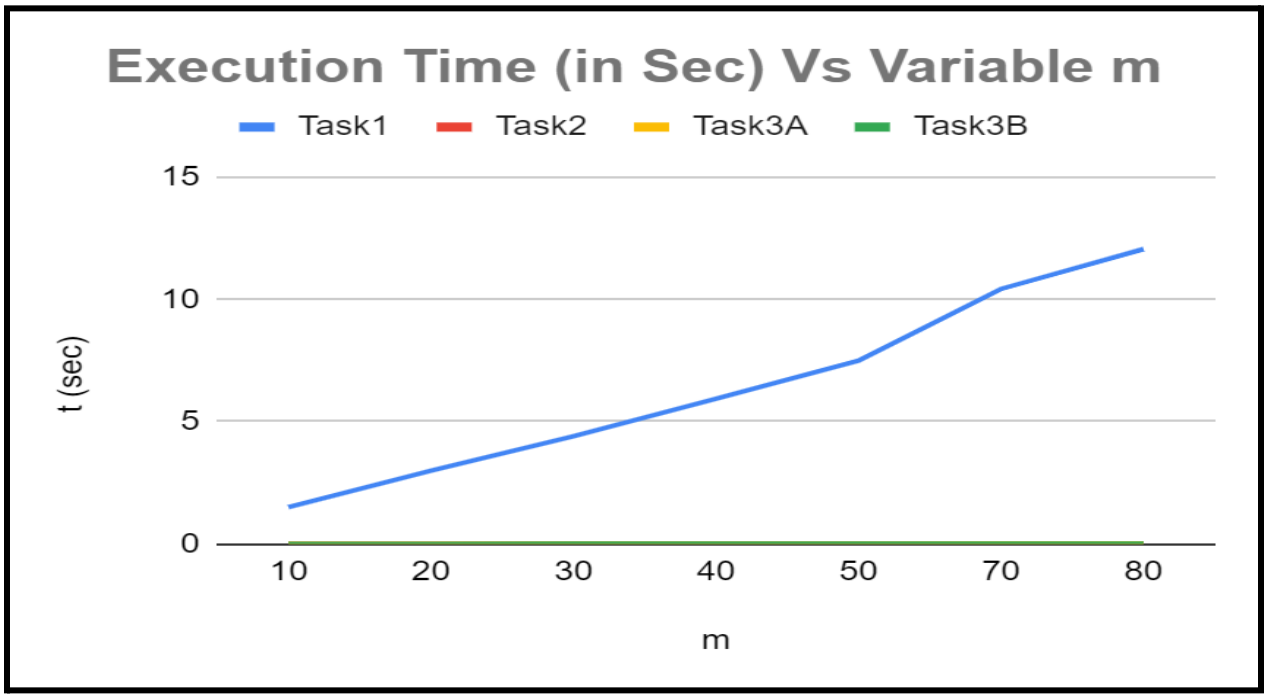


Table 2 : Execution Time (in Sec) Vs Input Size for Tasks 1 - 3b taking **n=5000**

m	Task1	Task2	Task3A	Task3B
10	1.505699	0.000606	0.002748	0.001955
20	2.993572	0.001256	0.005249	0.004665
30	4.404671	0.001888	0.008003	0.005949
40	5.94536	0.002472	0.01036	0.007697
50	7.502453	0.003228	0.014121	0.009251
70	10.431204	0.004481	0.018388	0.013446
80	12.05984	0.010083	0.020858	0.015709

Graph 2.1 : Execution Time (in Sec) Vs Input Size for Tasks 1- 3b taking **n=5000**



Because the execution time values in the preceding graph are overshadowing, we generated another graph to compare Task 2, Task 3a, and Task 3b.

Graph 2.2 : Execution Time (in Sec) Vs Input Size for Tasks 2- 3b taking **n=5000**

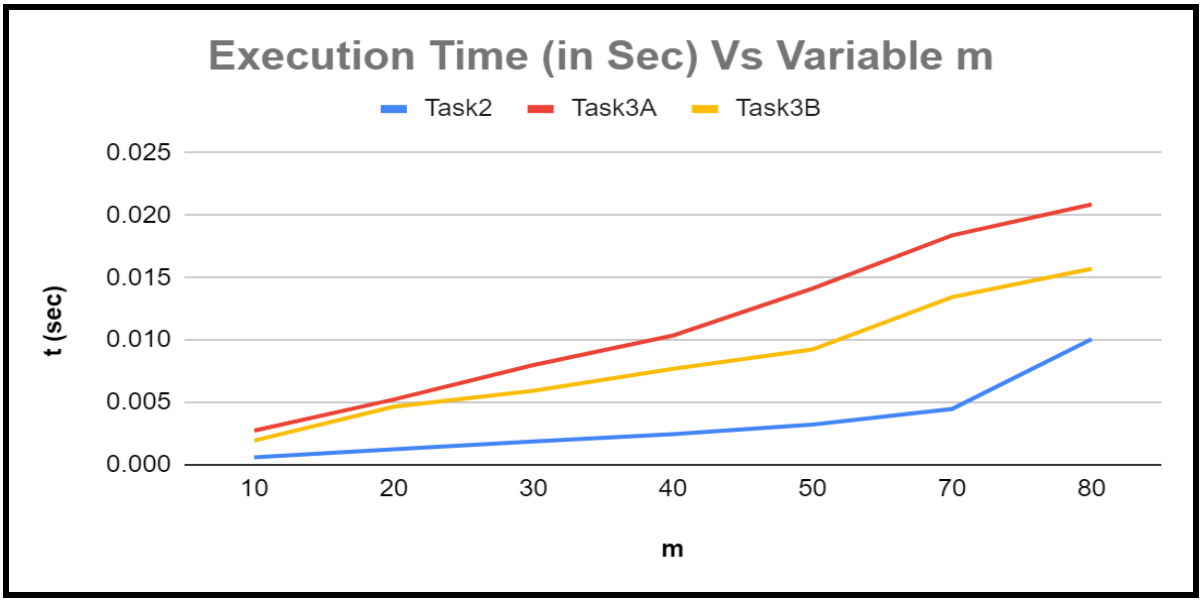


Table 3 : Execution Time (in Sec) Vs Input Size for Task 4 taking **m = 8 and k =4**

n	Task 4
4	0.007576
8	0.00712757
12	0.003965985
16	2.2454871
20	9.7543866
24	Infinity

Graph 3.1 : Execution Time (in Sec) Vs Input Size for Task 4 taking **m = 8 and k=4**

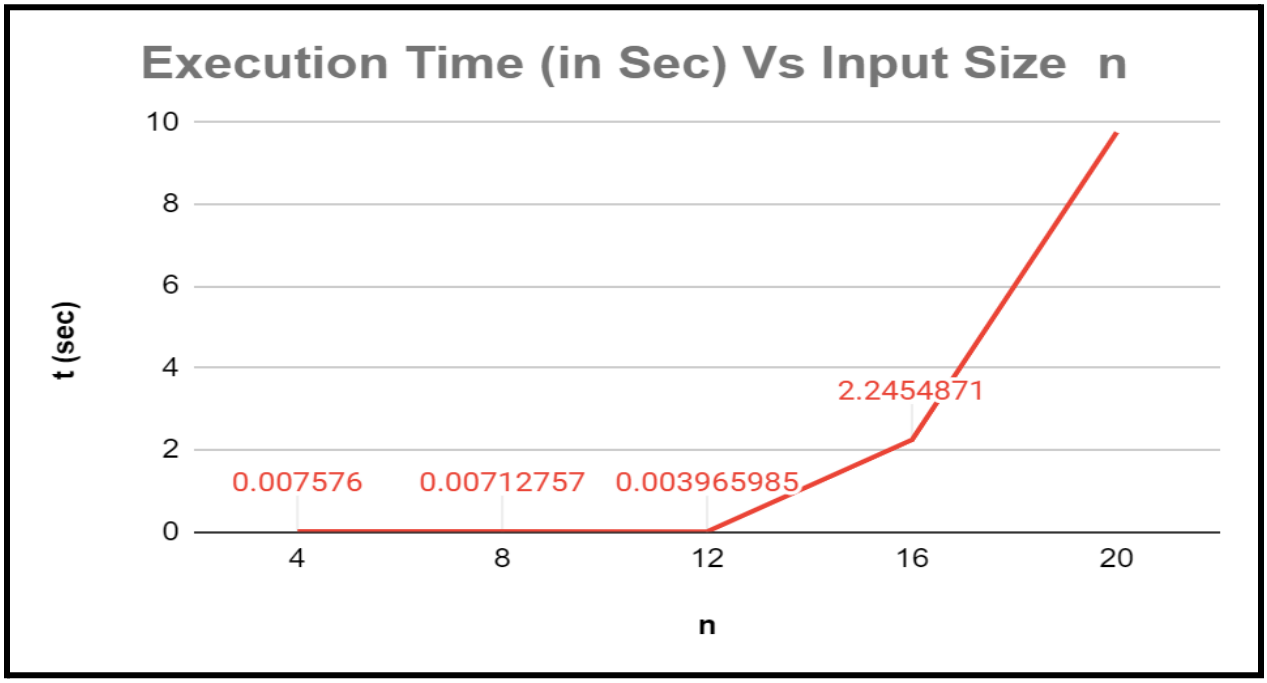
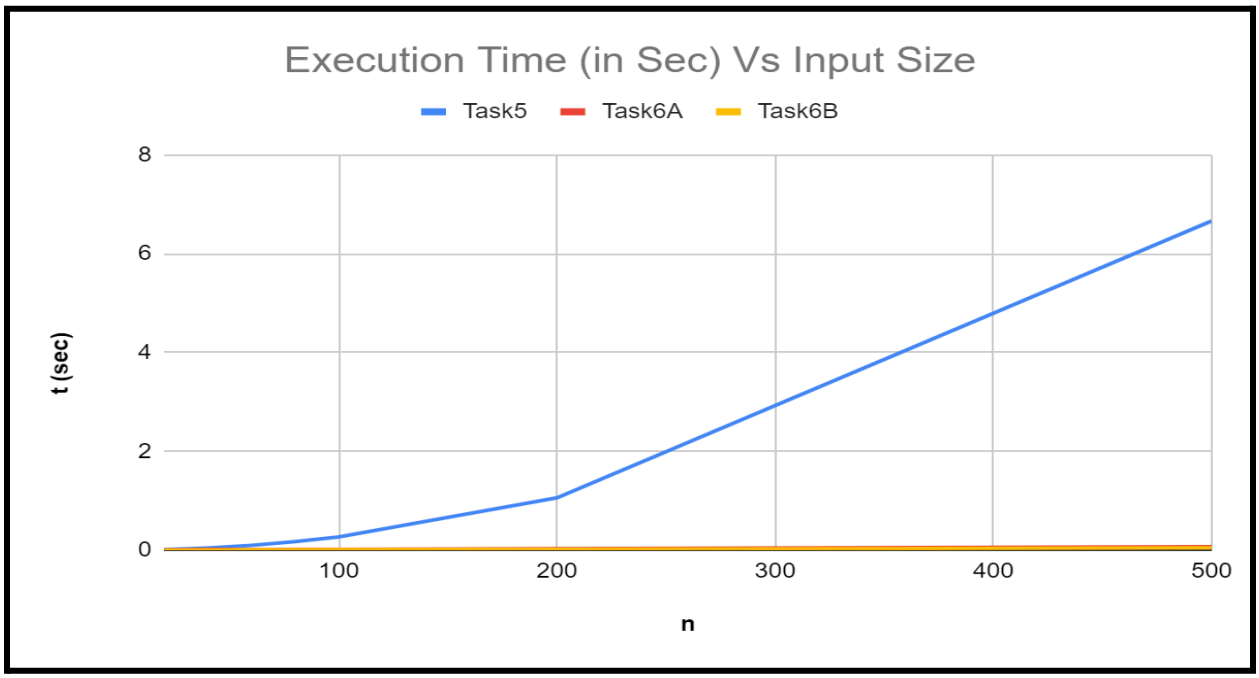


Table 4 : Execution Time (in Sec) Vs Input Size for Tasks 4- 6b taking **m=70 and k=40**

n	Task4	Task5	Task6A	Task6B
20	Infinity	0.010873	0.002709	0.001751
40	Infinity	0.041954	0.014764	0.003717
60	Infinity	0.097262	0.008722	0.005883
80	Infinity	0.171001	0.011492	0.007766
100	Infinity	0.266304	0.014647	0.009633
200	Infinity	1.059485	0.028305	0.018257
500	Infinity	6.663005	0.070461	0.045932

Graph 4.1 : Execution Time (in Sec) Vs Input Size for Tasks 5- 6b taking **m=70 and k=40**



Because the execution time values in the preceding graph are overshadowing, we generated another graph to compare Tasks 6a and 6b.

Graph 4.2 : Execution Time (in Sec) Vs Input Size for Tasks 6a and 6b taking **m=70 and k=40**

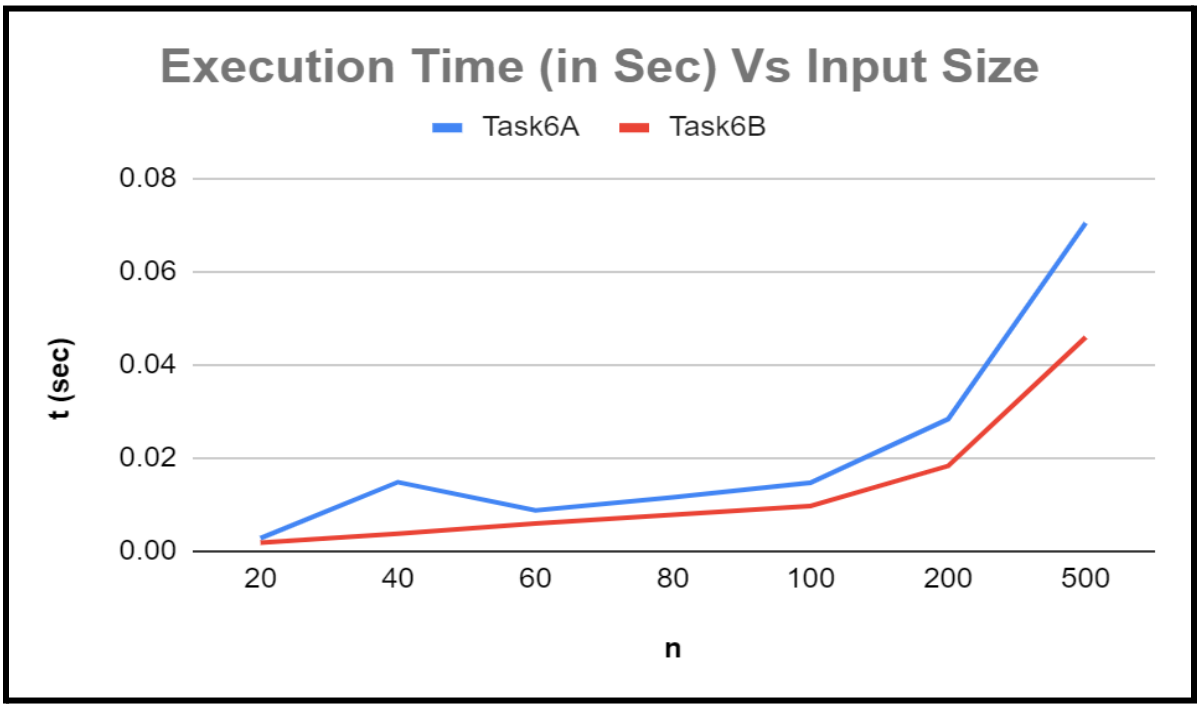
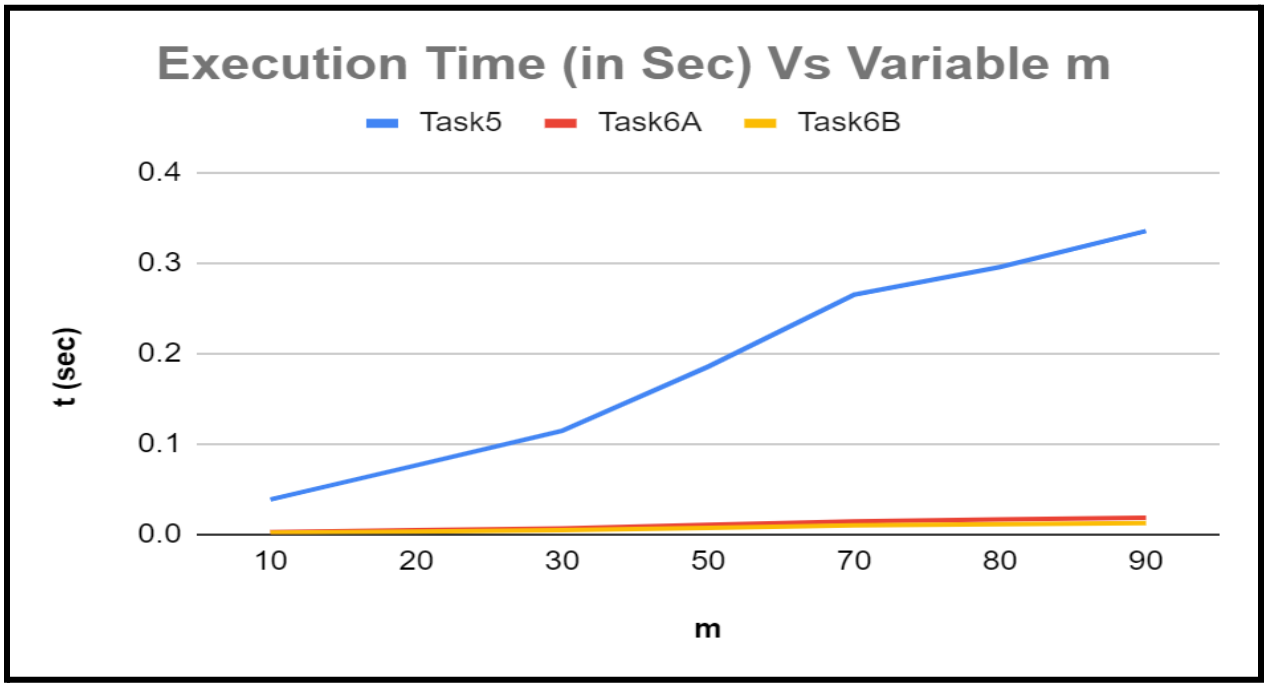


Table 5 : Execution Time (in Sec) Vs Input Size for Tasks 4- 6b taking **n=100 and k=40**

m	Task4	Task5	Task6A	Task6B
10	Infinity	0.038247	0.002277	0.001622
20	Infinity	0.076253	0.004358	0.002842
30	Infinity	0.11433	0.006345	0.004398
50	Infinity	0.185334	0.010181	0.00686
70	Infinity	0.264971	0.014166	0.009689
80	Infinity	0.295554	0.016329	0.010841
90	Infinity	0.335442	0.018131	0.012057

Graph 5.1 : Execution Time (in Sec) Vs Input Size for Tasks 5- 6b taking taking **n=100 and k=40**



Because the execution time values in the preceding graph are overshadowing, we generated another graph to compare Tasks 6a and 6b.

Graph 5.2 : Execution Time (in Sec) Vs Input Size for Tasks 6a and 6b taking taking **n=100 and k=40**

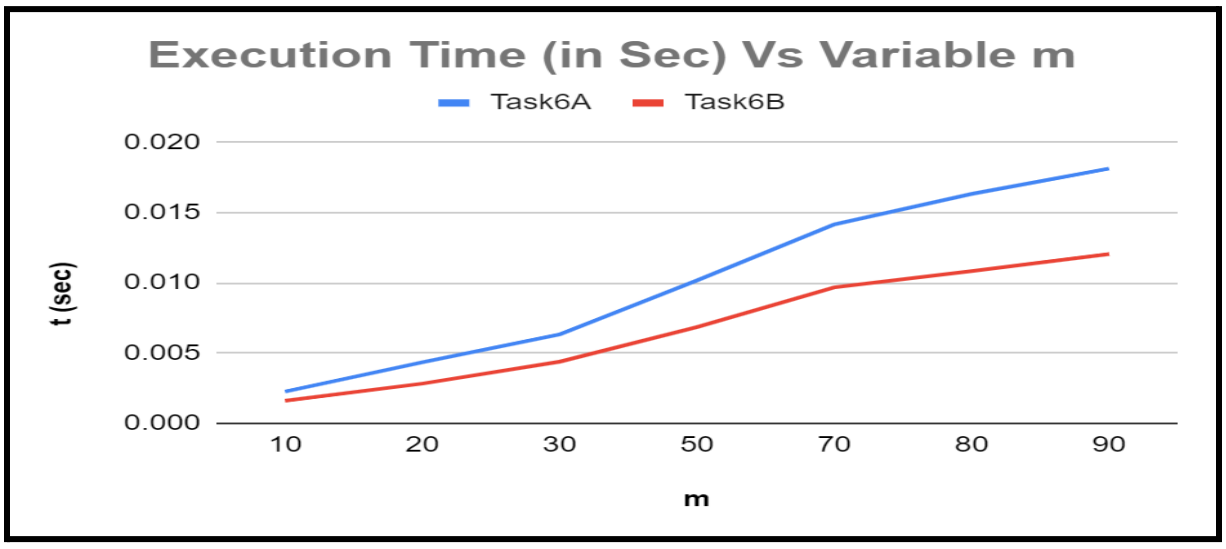
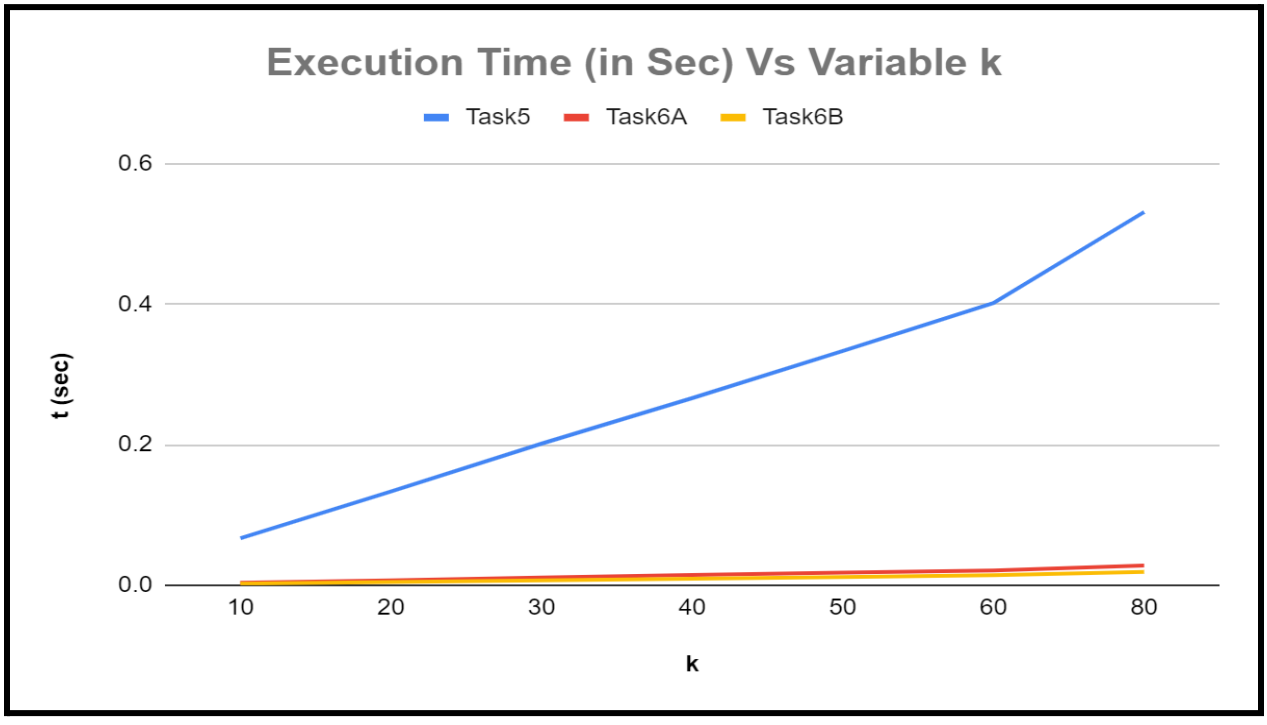


Table 6 : Execution Time (in Sec) Vs Input Size for Tasks 4- 6b **taking n=100, m=70**

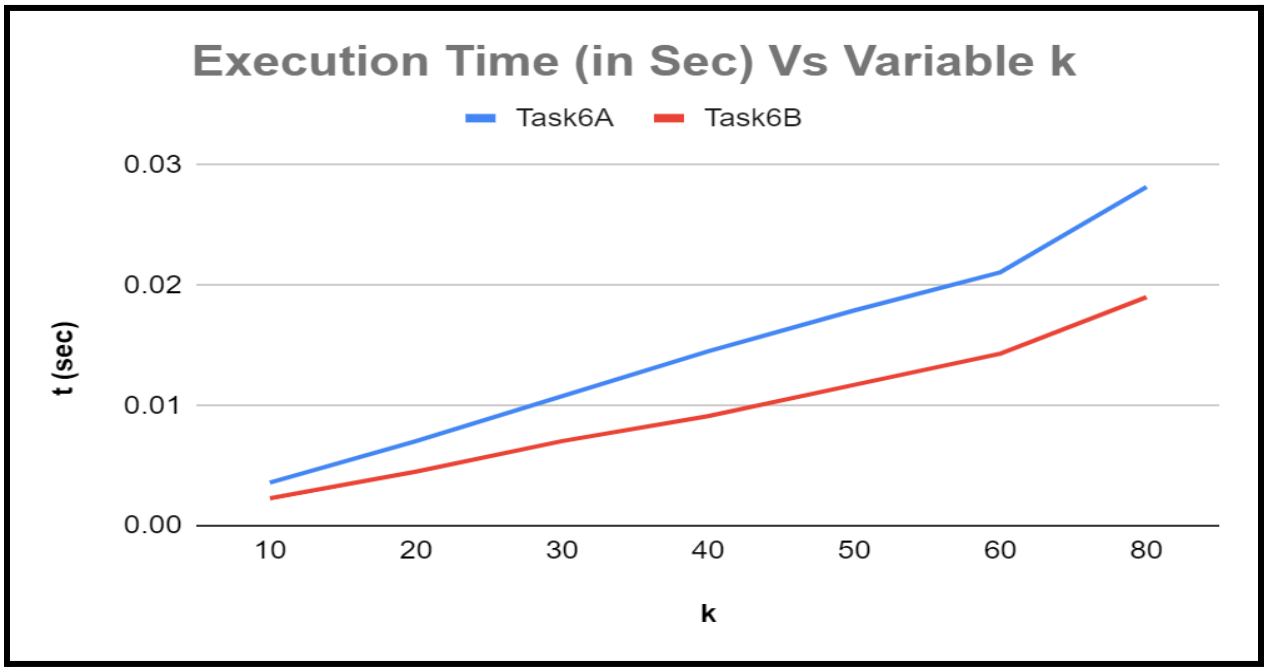
k	Task4	Task5	Task6A	Task6B
10	Infinity	0.067053	0.003611	0.0023
20	Infinity	0.133388	0.00705	0.004518
30	Infinity	0.201494	0.010795	0.007062
40	Infinity	0.266382	0.014531	0.00914
50	Infinity	0.333331	0.017927	0.011745
60	Infinity	0.401609	0.021105	0.014333
80	Infinity	0.530966	0.028206	0.019034

Graph 6.1 : Execution Time (in Sec) Vs Input Size for Tasks 5- 6b taking taking **taking n=100, m=70**



Because the execution time values in the preceding graph are overshadowing, we generated another graph to compare Tasks 6a and 6b.

Graph 6.2 : Execution Time (in Sec) Vs Input Size for Tasks 6a and 6b taking taking **taking n=100, m=70**



Bonus Question:

Problem3: Given a matrix A of m × n integers (non-negative) representing the predicted prices of m stocks for n days and an integer c (positive), find the maximum profit with no restriction on the number of transactions. However, you cannot buy any stock for c days after selling any stock. If you sell a stock on day i, you are not allowed to buy any stock until day (i + c + 1).

Alg7: Θ(m*2^n) complexity with Brute Force algorithm for solving Problem3

Design :

- We followed a recursive approach to exhaust all possible combinations of transactions across all stocks considering cooldown i.e. cannot buy a stock for cooldown number of days after selling on that day.
- Our base condition is returning 0 if n i.e. if day is 0.
- Otherwise, maxVal stores the returned value of a recursive call with parameter n-1.
- Then we iterate over all the stocks in the outer loop from 0 to m. Nesting another loop in it which iterates over n-1 to 0.
- Here we take the maximum of previously computed maxVal and return value of the recursive call made with parameter j-cooldown-1 (where j is the iterating variable of the nested loop) added to the price difference between that stock on n th day and j th day.
- Finally returning the maximum profit achieved i.e. maxVal until that day. Simultaneously filling a table to later backtrack for exact transactions responsible for the achieved maximum profit.

Pseudo Code:

```
Task7(m,n,c,p)
        M[n] <- -1
        return brute(n-1)
brute(n)
        if(n==0)
                return 0;
        maxVal = brute(n-1)
        For s = 0 to m
                For j = n-1 down to 0
                if(i-c-1 >=0)
                        maxVal = max(maxVal, brute(j-c-1)+p[s][n] - p[s][j]
                Else
                        maxVal = max(maxVal, p[s][n] - p[s][j])
        Return M[n] = maxVal
```

Time and Space Complexity:

- The time complexity for this brute algorithm is $\Theta(m*2^n)$ as we are generating a recursion tree of height 2^n and work done at each level is m. Hence giving us the complexity stated above.
- The space complexity of this algorithm is O(n*m) as every recursion call needs memory in the stack.

Proof of Correctness:

Brute force algorithm we implemented finds the maximum profit by exhaustion. All possible cases where we can make as many transactions as possible which are

non-overlapping across all stocks are checked. Hence consistently giving the maximum profit along with the transactions.

Alg8: Θ(m*n^2) complexity with Dynamic Programming Algorithm for solving Problem3

Design :

- In this algorithm we construct the sub-problem to be M[i] which is defined as the maximum profit achieved until i th day by performing as many transactions as possible considering transactions across all stocks.
- Our goal is to compute M[n-1] i.e. maximum profit achieved until the last day by performing as many transactions as possible across all stocks.

  GOAL => Compute M[n-1]
- Base cases can be described as when i is equal to zero then M[i] is assigned zero.
- If the recursion parameter i.e. i - cooldown - 1 <
- Otherwise, the value for M[i] is maximum of previously computed value i.e. M[i-1] & maximum profit achieved across all stocks until i-cooldown-1 th day (stock price of s th stock at i th day - stock price of s th stock at b th day where s & b are iterated through 1 to m and 1 to i-1 respectively) added to M[b-cooldown-1]. (Refer the Mathematical Formulation)
- After achieving all the values for M[n-1], the corresponding values can be stored in the Memoization table. This table can be used to back track the transactions responsible for the profit achieved at that position.
- This can be done using the BackTracking logic described in the pseudo code. Hence obtaining the exact transactions that return maximum profit.

Mathematical Formulation :

$M[i]$

$Case\ 1$: 0 $\quad if\ i\ =\ 0$

$Case\ 2$: $otherwise$

$max(M[i-1],\ M[b-c-1] + p[s][i] - p[s][b])$ $\quad if(b-c-1 >= 0)$

$max(M[i-1],\ p[s][i] - p[s][b])$ $\quad if(b-c-1 < 0)$

$where\ 1 <=\ s\ <=\ m\ and\ 1 <=\ b\ <=\ n-1$

Pseudo Code:

```
Task8(m,n,c,M)
        M[0] = 0
        For d = 1 to n
                maxVal = M[d-1]
                For s = 0 to m
                        For j - d-1 down to 0
                                if(j-c-1 >= 0)
                                        maxVal = max(maxVal, M[j-c-1] + p[s][d] - p [s][j]
                                Else
                                        maxVal = max(maxVal, p[s][d]-p[s][j])
                M[d] = maxVal

BackTracking(m,n,c, M)
d=n-1
while(true)
        if(d <= 0)
                Break
        if(M[n] == M[n-1])
                D = d-1
        Else
                For s = 0 to m
                        For j = d-1 down to 0
                                if(j-c-1 >= 0)
                                        if(M[d] == M[j-c-1] + p[s][d] - p[s][j])
                                                Sol U {s,j,d}
```

d=j-c-1
Break

Else

if(M[d] == p[s][d] - p[s][j])
Sol U {s,j,d}
d= j-c-1
Break

Time and Space Complexity:

- The time complexity of this algorithm is Θ(m*n^2) as we are filling the DP table of size n and to fill each element the work done is m*n.
- The space complexity of this algorithm is O(n). As we need to store the dp table of size n.

Proof of Correctness:

Here we consider our subproblem to be M[i] which can be defined as maximum profit achieved until i th day and performing at as many transactions as possible considering all the stocks. The base case for [0] is 0 vacuously(i.e. If i = 0). Basically, as we proceed taking a single step down in the recursion tree, at least one of i or j is decrementing by 1. Regardless of considering whatsoever child of subproblems, we eventually hit the base case. This means that if we use those subproblems as our base cases, we can make sure that every branch of the induction tree ends at a base case where all of the assumptions are met. We fill the table based on both previous optimal values and optimally calculating at that level. Hence by definition of the subproblem M[n] should produce the maximum profit for the whole. So by property of optimal substructure, our solution is bound to be correct.

Alg9: Θ(m*n) complexity with Dynamic Programming algorithm for solving Problem3

Design :

- In this algorithm we construct the sub-problem to be M[i] which is defined as the maximum profit achieved until i th day by performing as many transactions as possible considering transactions across all stocks.
- Our goal is to compute M[n-1] i.e. maximum profit achieved until the last day by performing as many transactions as possible across all stocks.

  GOAL => Compute M[n-1]
- Base cases can be described as when i is equal to zero then M[i] is assigned zero.
- If the recursion parameter i.e. i - cooldown - 1
- Otherwise, the value for M[i] is maximum of previously computed value i.e. M[i-1] & maximum profit achieved across all stocks until i-cooldown-1 th day (stock price of s th stock at i th day - stock price of s th stock at b th day where s & b are iterated through 1 to m and 1 to i-1 respectively) added to M[b-cooldown-1]. (Refer the Mathematical Formulation)
- Here to reduce the repetitive checking, we introduce a new variable from the start i.e maxDiff. Here if we already take the maximum of differences between M[a] and stock price[m][a] at every i th day iteration. We can eliminate the nested loop that iterates through 1 to i-1. Hence scaling down the time complexity from n^2 to just n.(Refer the Mathematical Formulation)
- After achieving all the values for M[n-1], the corresponding values can be stored in the Memoization table. This table can be used to back track the transactions responsible for the profit achieved at that position.
- This can be done using the BackTracking logic described in the pseudo code. Hence obtaining the exact transactions that return maximum profit.

Mathematical Formulation :

$M[i]$

$Case\ 1:\ 0 \qquad if\ i\ =\ 0$

$Case\ 2:\ max(M[i\ -\ 1],\ maxDiff[s]\ +\ p[s,i] \qquad otherwise$

$if(i\ -\ c\ -\ 1\ >=\ 0)$

$maxDiff[s]\ =\ max(maxDiff[s], M[i\ -\ c\ -\ 1]\ -\ p[s,i]$

$else$

$maxDiff[s]\ =\ max(maxDiff[s],\ -\ p[s,i]$

$where\ 1\ <=\ s\ <=\ m$

Pseudo Code:

```
Task9A(m,n,c)
    M[n] <- -1
    D[m] <- 0
    For i = 0 to m
        D[i] = -p[i][0]
    Return mCompute(n-1, D, M)

mCompute(n, D,M)
    if(n==0)
        Return 0
    if(M[n] is uninitialized)
        maxVal = max(n-1, D, M)
        For s =0 to m
            maxVal = max(maxVal, D[s] + p[s][n]
            if(n - c - 1 >=0)
                D[s] = max(D[s], mCompute(n-c-1, D, M) - p[s][n])
            Else
```

```
                    D[s] = max(D[s], -p[s][n])
            Return M[n] = maxVal
        Return M[n]

Task9b(m, n, c,p)
        M[n] <- -1
        M[0] <-0
        D[m] <- 0
        For i = 0 to m
            D[i] = -p[i][0]
        For d = 1 to m
            maxVal = M[d-1]
            For s=0 to m
                maxVal = max(maxVal, D[s] + p[s][d]
                if( d - c -1 >= 0)
                    D[s] = max(D[s], M[d-c-1] - p[s][d])
                Else
                    D[s] = max(D[s], -p[s][d]
            M[d] = maxVal

BackTracking(m,n,c, M)
d=n-1
while(true)
        if(d <= 0)
            Break
        if(M[n] == M[n-1])
            D = d-1
        Else
            For s = 0 to m
```

```
For j = d-1 down to 0
    if(j-c-1 >= 0)
        if(M[d] == M[j-c-1] + p[s][d] - p[s][j])
            Sol U {s,j,d}
            d=j-c-1
            Break
    Else
        if(M[d] == p[s][d] - p[s][j])
            Sol U {s,j,d}
            d= j-c-1
            Break
```

Time and Space Complexity:

- The time complexity of this algorithm is Θ(m*n).We need to fill the 2D dp table of size n but, as we are eliminating the nested inner loop which iterates to n at worst case. The work done to fill each element is m.
- The space complexity of this algorithm is O(n). As we need to store the dp table of size n.

Proof of Correctness:

Here we consider our subproblem to be M[i] which can be defined as maximum profit achieved until i th day and performing at as many transactions as possible considering all the stocks. The base case for [0] is 0 vacuously(i.e. If i = 0). Basically, as we proceed taking a single step down in the recursion tree, at least decrementing i by 1. Regardless of considering whatsoever child of any subproblem, we eventually hit the base case. This means that if we use those subproblems as our base cases, we can make sure that every

branch of the induction tree ends at a base case where all of the assumptions are met. We fill the table based on both previous optimal values and optimally calculating at that level. Hence by definition of the subproblem M[n] should produce the maximum profit for the whole. So by property of optimal substructure, our solution is bound to be correct.

Experimental Comparative Study:

We analyzed the algorithm's execution time and made comparison graphs using various input sizes and randomly generated inputs. The performance tables and graphs are shown below.

Table 7 : Execution Time (in Sec) Vs Input Size for Tasks 7 **taking m = 8 and c=4**

n	Task 7
20	0.35495
25	1.965344
30	31.655097
35	515.05298
40	Infinity

Graph 7.1 : Execution Time (in Sec) Vs Input Size for Tasks 7 **taking m = 8 and c= 4**

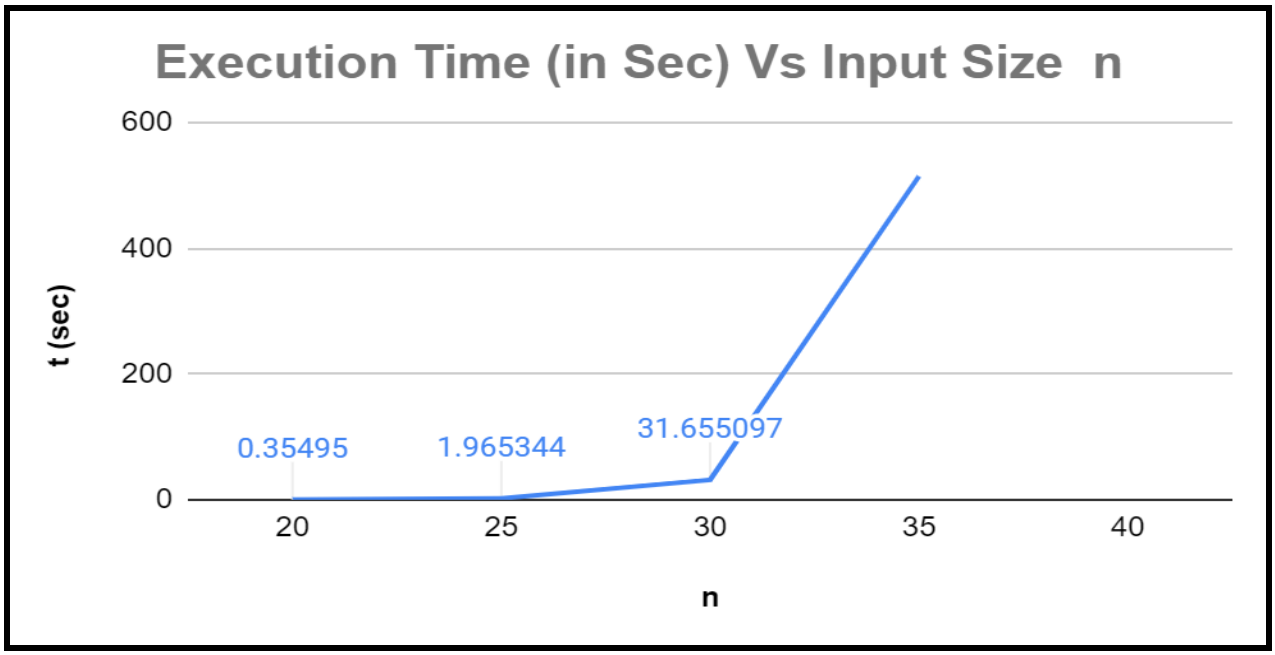
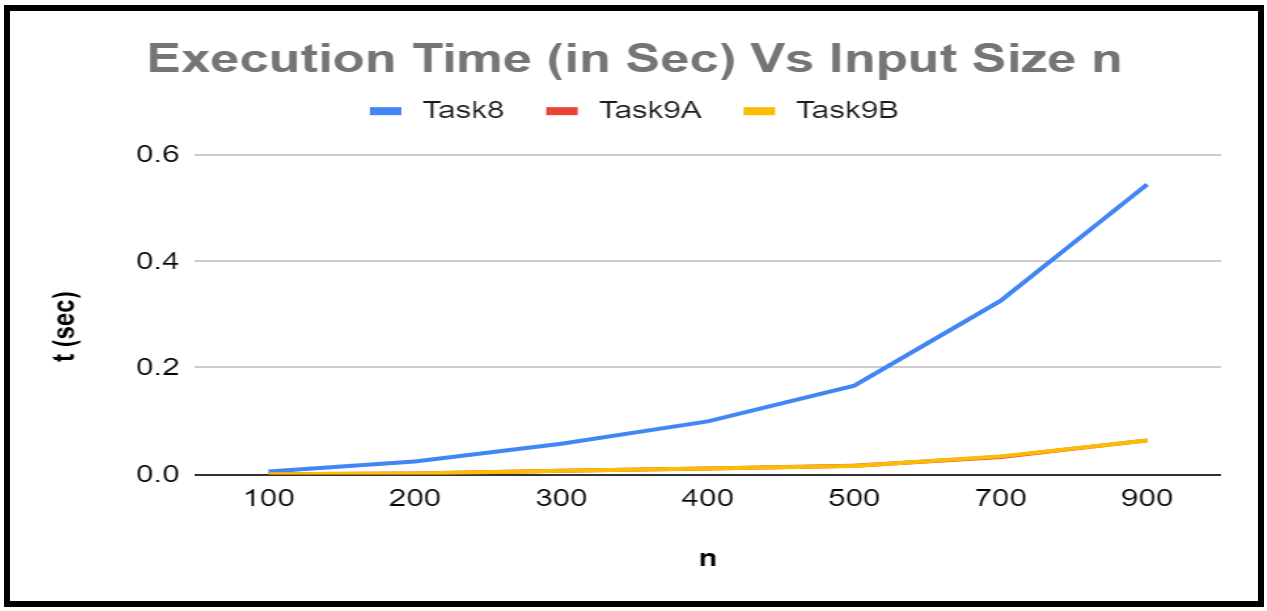


Table 8 : Execution Time (in Sec) Vs Input Size for Tasks 7- 9b **taking m=70 and c=50**

n	Task7	Task8	Task9A	Task9B
100	16.965185	0.005672	0.0005	0.000497
200	Infinity	0.024364	0.001904	0.00185
300	Infinity	0.0576	0.006824	0.007033
400	Infinity	0.099612	0.011265	0.011053
500	Infinity	0.166655	0.016191	0.015846
700	Infinity	0.326189	0.032897	0.034015
900	Infinity	0.54473	0.064059	0.063976

Graph 8.1 : Execution Time (in Sec) Vs Input Size for Tasks 8- 9b **taking m=70 and c=50**



Because the execution time values in the preceding graph are overshadowing, we generated another graph to compare Tasks 9a and 9b.

Graph 8.2 : Execution Time (in Sec) Vs Input Size for Tasks 9a and 9b **taking m=70 and c=50**

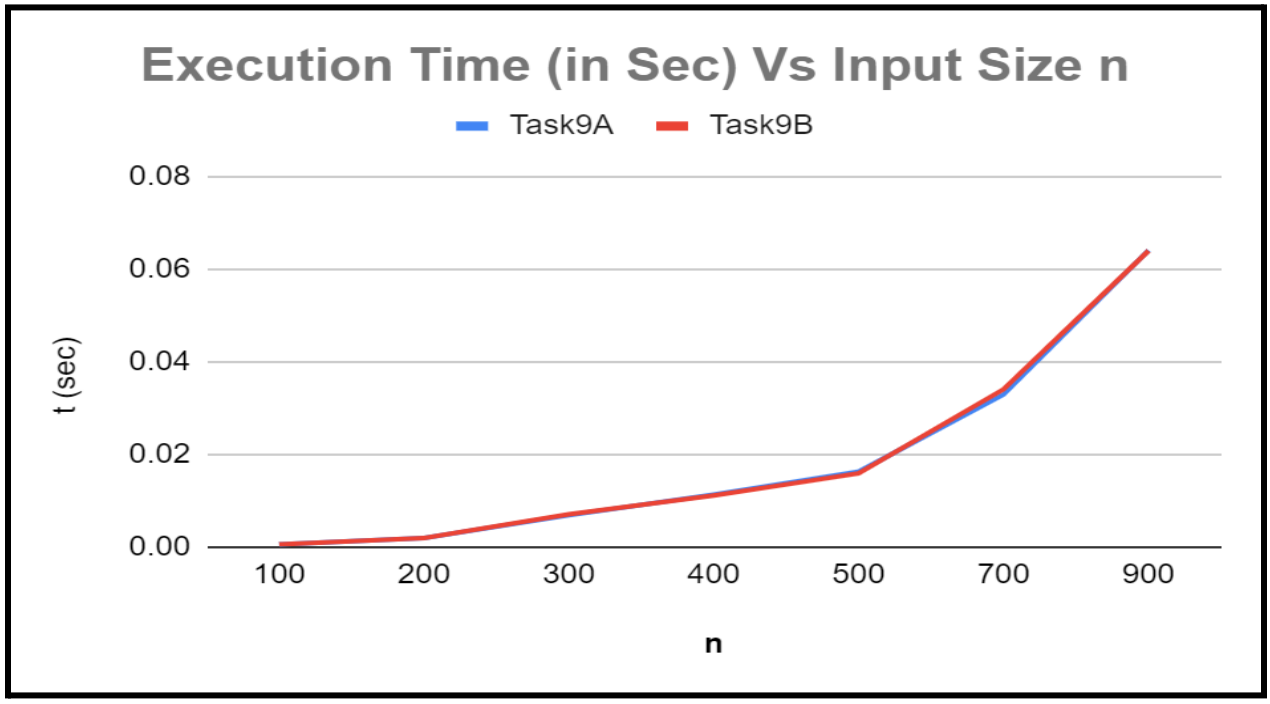
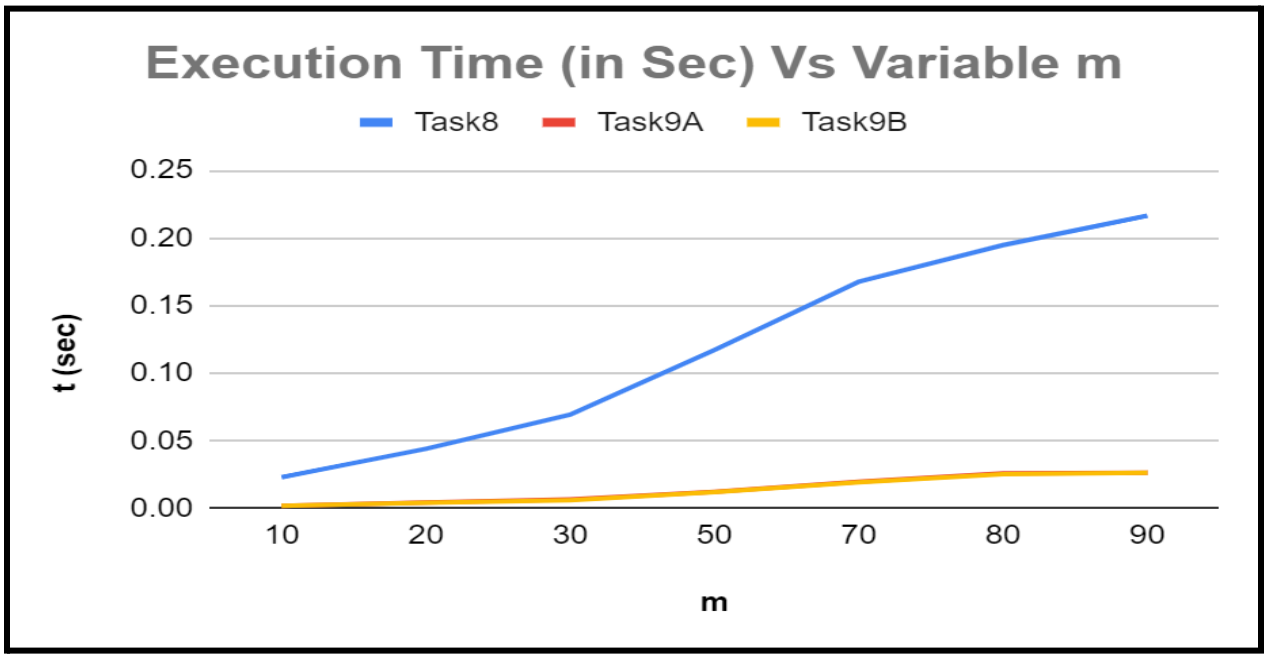


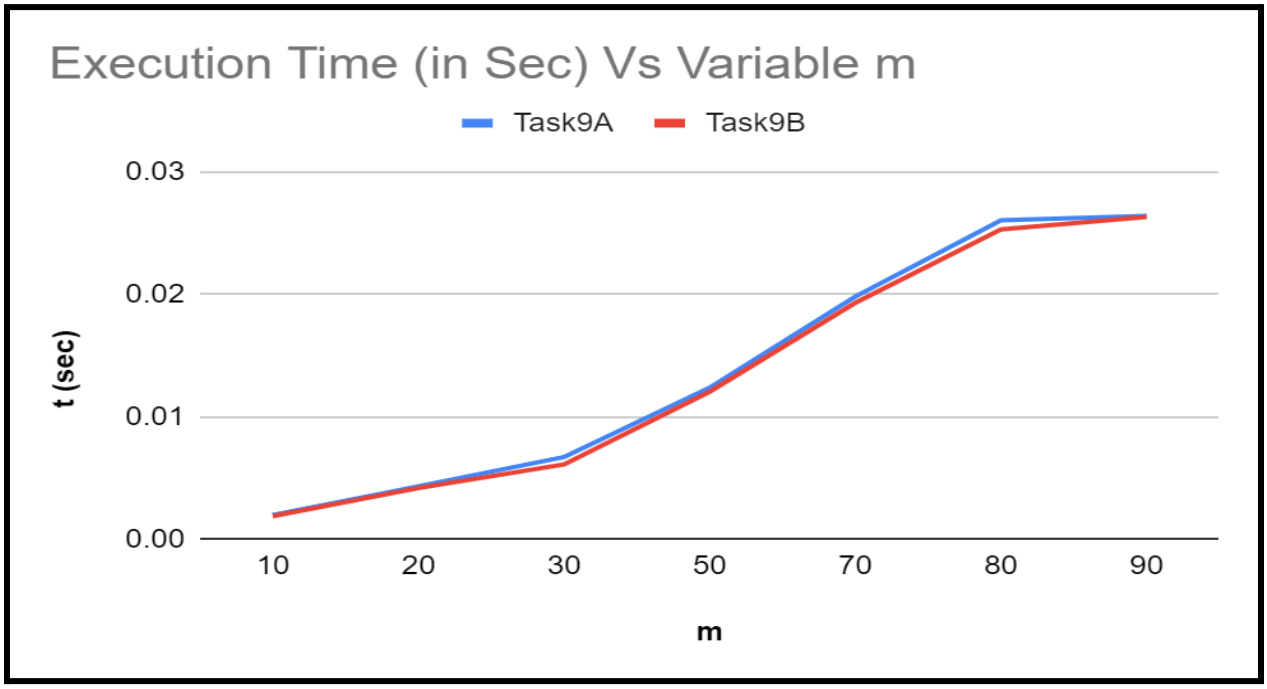
Table 9 : Execution Time (in Sec) Vs Input Size for Tasks 7- 9b **taking n=500 and c=50**

m	Task7	Task8	Task9A	Task9B
10	Infinity	0.023117	0.002001	0.001914
20	Infinity	0.044108	0.004336	0.004214
30	Infinity	0.069367	0.006742	0.006134
50	Infinity	0.117195	0.012394	0.012059
70	Infinity	0.167721	0.019825	0.019326
80	Infinity	0.194842	0.026049	0.025316
90	Infinity	0.216621	0.026423	0.026329

Graph 9.1 : Execution Time (in Sec) Vs Input Size for Tasks 8- 9b **taking n=500 and c=50**



Graph 9.2 : Execution Time (in Sec) Vs Input Size for Tasks 9a and 9b **taking n=500 and c = 50**



Conclusion

Problem 1:

- The most optimal solution in problem1 will be the Greedy algorithm, even though both greedy and DP had a time complexity of Θ(mn). We can clearly observe this in plots 1 & 2 shown above, that greedy is optimal for both smaller and larger inputs.
- To compare with bruteforce, it doesn't need much of an intuition, as we are considering all possible cases to compute the maxprofit, this will be the slowest of all.

Problem 2:

- It is clear that DP algorithm is faster in problem2 and as we cannot develop a greedy algorithm for this, we can safely conclude that DP algorithm is optimal in both space and time.
- Here, getting the idea to maintain a 2D maxdiff table in task6A for memoization recursion is crucial because the same can be done with 1D maxdiff table in task 6B.
- As an observation, we can reduce the space complexity furthermore in the DP implementation here. There is no need to fill the upper triangular matrix of the dp table which will save a lot of space for larger inputs. This is observed while developing the backtracking algorithm for these dp algorithms, where this kind of optimization is observed.

Problem3:

- The same argument can be applied here as well: bruteforce is very slow for larger numbers of n, where it goes to infinity, which is evidently observed in the plots shown above.
- Here the critical idea is to use the same recursion relation described in problem2 with some modifications based on the cooldown period.
- Dividing the dp table filling into two different conditions based on the availability of that index in the dp table is very important. Due to this idea, we were easily able to incorporate the same recursion relations used in problem 2.