COT5405 Algorithms Programming Project

PROJECT REPORT

Submitted for the course: Analysis of Algorithms (COT5405)

By

Rohan Paranjpe	**94887293**
Anol Kurian Vadakkeparampil	**56268544**
Rashi Pandey	**88124446**

Table of Contents

Chapter	Topic	Page no.
1	Team members and contribution	3
2	Design and Analysis of Algorithms	4
3	Experimental Comparative Study	24
4	Conclusion	27

Team members and contribution

Rohan Paranjpe	**Task 3(a) and 6(b)**
Anol Kurian Vadakkeparampil	**Task 1, Task 2, Task 5, Task 9(b)**
Rashi Pandey	**Task 3(b), Task 4, Task 8,**

Design and Analysis of Algorithms

Defining the Scenario:

We are provided with a 2d array with price predictions for m stocks for n consecutive days. At day j, price for stock i is given by A[i][j] with i ranging from 1 to m and j ranging from 1 to n. The task is to find maximum possible profit by buying and selling stocks. Price of stock at any day is a non-negative integer. We are allowed to hold only one share of one stock at a time and we can buy a stock on the same day we sell another stock.

Problem Statement 1:

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

Alg1

Design a Θ(m*n^2) time brute force algorithm for solving Problem1

Alg 1 design:

- We define a function named maxProfitBruteForce with the formal parameters passed as the 2d array depicting the prices of m stocks for n days, m depicting the number of stocks and n as the number of days.
- We initially initialize a variable max_diff (representing the maximum profit by calculating the difference between the prices of consecutive days for a given stock) as 0.
- The variables dayS, dayB and stock are defined to represent the sell day, buy day and the stock number respectively.
- An outer loop k runs for the number of stocks from 0 to less than m.
- An inner loop is defined to run for the stock price for each day depicted by i and the innermost loop j is defined inside this loop, starting from i+1 to the end, to check for the maximum difference from each further day from the ith day.
- If the difference between A[k][j] and A[k][i] is greater than max_diff, then update max_diff with this difference.
- Inside the if condition, everytime we get a new maximum difference, update the variables stock as k, dayS as j and dayB as i.
- We print the output in the required format with incrementing the values of dayS, dayB and stock by 1.
- The execution time is shown by defining the t1 (before the start) and t2 (after the code) variables using the function System.nanoTime() and printing their difference.

Pseudocode:

```
maxProfitBruteForce(int A[][], int m, int n)
Maintain a variable max_diff for the maximum profit initialized to 0
Maintain the variables dayS, dayB and stock initialized to -1
For k=0,1,....,m-1
  For i=0,1,....,n-1
    For j=i+1,i+2,....,n-1
      If the difference A[k][j]-A[k][i] is greater than max_diff then
        Update max_diff=A[k][j]-A[k][i]
        Update stock=k
        Update dayS=j
        Update dayB=i
      Endif
    Endfor
  Endfor
Endfor
Return the maximum profit max_diff
End
```

Proof of Correctness:

We can prove the correctness using proof by induction using loop invariant. We assume that the array has non-null values and contains at least 1 element. Suppose, the array contains just a single element, and as we have initially initialized the max_diff to 0, when the size of the array is 0, the maximum profit is 0 as there is no buy and sell transaction for the first day. At the start of each iteration, if max_diff is present in the array, then it is present in the subarray from A[k][i] to A[k][n-1].
For i =0, max_diff is 0, and it is present in the subarray A[k][0] to A[k][n-1].
As the i increments, j also increments from i+1, and we check for the maximum profit, max_diff is present in the subarray from A[k][i+1] to A[k][n-1], therefore it is true for i+1.
By inductive hypothesis, when the loop terminates, the max_profit will be present in the array and that is correctly returned by our algorithm.

Time Complexity:

As there are three nested loops with the outermost loop k running from from 0 to m-1, the inner loop i from 0 to n-1 and the innermost loop j from i+1 to n-1, the total running time is O(m*n*n). Therefore, the worst case time complexity is O(m*n^2).

Space Complexity:

As we are not using extra space ,so the space complexity is O(1).

<u>Alg2</u>

Design a Θ(m∗n) time greedy algorithm for solving Problem statement 1

Alg 2 design:

- We define a function named maxDiffGreedy with the formal parameters passed as the 2d array depicting the prices of m stocks for n days, m depicting the number of stocks and n as the number of days.
- We initially initialize a variable maxProfit (representing the maximum among all profits by calculating the difference between the price at the current pointer and the cheapest price so far for a given stock) as 0.
- We declare an array cheapestDayArray to store the cheapest price day,i.e. the buy day for each stock.
- The variables cheapestDay, maxProfitDay,stock and cheapest(initialized to the first element of first row of 2d array) are defined to represent the buy day, sell day, the stock number and the cheapest price up to date respectively.
- An outer loop i runs for the number of stocks from 0 to less than m.
- An inner loop j runs from 1 to n-1.
- We scan from left to right of the array calculating the maximum profit and also the cheapest price encountered yet.
- If we come across a cheaper price than the current cheapest price , we update the new cheapest price.
- If the price value A[i][j] is greater than the cheapest price so far and if we come across a profit greater than the last maximum profit, then we update the new maximum profit.
- Print the required output consisting of buyday, sellday, stock number and maximum profit.
- The execution time is shown by defining the t1 (before the start) and t2 (after the code) variables using the function System.nanoTime() and printing their difference.

Pseudocode:

```
maxDiffGreedy(int A[][], int m, int n)
Maintain a variable maxProfit for the maximum profit initialized to 0
Maintain the variables cheapestDay, maxProfitDay and stock initialized to -1
Declare an array cheapestDayArray with the size as number of stocks m
For i=0,1,....,m-1
  For j=1,....,n-1;Maintain a variable cheapest initialized to A[i][0]
    If cheapest is greater than A[i][j] then
       Update cheapest to A[i][j]
       Update cheapestDayArray[i] to J
    Endif
    If A[i][j] is greater than cheapest then
       Update   maxProfit   to   maximum   between   the   old   maxProfit   and
A[i][j]-cheapest
       Update stock tot the current stock i
       Update maxProfitDay to j
       Update cheapestDay to cheapestDayArray[i]
    Endif
 Endfor
Endfor
Return maxProfit
End
```

Proof of Correctness:

We need to show that our greedy algorithm is feasible and optimal at all instants. We will prove that our greedy algorithm always “stays ahead” of the optimal solution. Let S be the solution output by the algorithm and O be the optimum solution.
If we assume that S is not equal to O, i.e., S is not optimal. But, in our algorithm, at each step, we are using the cheapest price so far and comparing it with the next available price in the array for a given stock. Also, the maximum profit is also the maximum encountered so far as we are always comparing it with the maximum profit so far and the difference of the current price of that day to the cheapest price seen so far. Therefore, there is a contradiction. So, our greedy algorithm yields an optimal solution satisfying all constraints at each step of the execution.

Time Complexity:

As there are two nested loops with the outer loop i running from from 0 to m-1 and the inner loop j from 1 to n-1, the total running time is O(m*n).

Space complexity:

The space complexity for the above algorithm is O(m) as we are using an extra array cheapestDayArray of the size of m to store the buy day for each stock.

<u>Alg3</u>

Design a Θ(m∗n) time dynamic programming algorithm for solving Problem statement 1

3.a) Design of recursive implementation of Alg3 using Memoization:

- We run a loop i for m stocks.
- We call the function named maxprofit with the formal parameters passed as the array depicting the prices of the i th stock for n days, m depicting the ith stock and n as the number of days-1.
- We use a separate array profit to store the results for future use as is done in the case of memoization.
- We initially initialize a variable maxtotal (representing the maximum profit) as 0.
- The variables tempCheapest,buy,stock and sell are defined to represent the buy day, the stock number and the sell day.
- A separate condition is set for the case of 2 days for a buy and sell transaction.
- We initially initialize a variable maxtotal (representing the maximum profit) as 0.
- The nth index of profit array is initialized with the maximum between 0 and the difference between the prices of the two days.
- Check is carried out for the maxtotal and it is updated to the new maximum profit.
- The buyday is updated using the above check.
- For n>2, The nth index of profit array is initialized with the maximum between 0 and the results obtained from the recursion in which we keep on passing by decrementing the number of days(n) each time and storing the result in profit[n].
- The results of the profit till the previous day are stored and used by adding it to the difference between the current and previous day.
- Check is carried out for the maxtotal and it is updated to the new maximum profit.
- The buyday is updated using the above check.
- The value of last index of the array profit is returned as the maximum profit following the top- down approach.

Pseudocode:

```
maxProfitRecMem(int A[]][], int m, int n)
For i=0,1,...,m-1
  maxprofit(p[i], n(length(A[i])-1), m(i))
  If n is equal to 1 then
    profit[n]= maximum(0,profit[n]=p[n]-p[n-1])
    If profit[n] is greater than maxtotal then
      buy=n-1
    Endif
    Return profit[n]
  Endif
  Else
    profit[n]=maximum(0,maxprofit(p,n-1,m)+p[n]-p[p-1])
    If profit[n] is equal to 0 then
      tempCheapest=n;
    Endif
    If profit[n] is greater than maxtotal then
      buy=tempCheapest
      sell=n
      stock=m
    Endif
    maxtotal=maximum(maxtotal,profit[n])
    Return profit[n]
  End
```

Recursive Formulation expressing optimal substructure property:

Opt(j)=max(0,Opt(j-1)+(p(j)-p(j-1)))
OR
OPT(*j*) = max(*0*, maxprofit(*j* − 1)+p(j)-p(j-1)) = maxprofit(*j*)

Proof of Correctness:

We will prove the correctness using proof by induction. By definition OPT(0) = 0. Now, when we take some *j* > 0, and suppose by induction that maxprofit(*i*) correctly computes OPT(*i*) for all *i* < *j*. By the induction hypothesis, we know that maxprofit(*p*(*j*)) = OPT(*p*(*j*)) and maxprofit(*j* − 1) = OPT(*j* − 1); and therefore it follows that:

OPT(*j*) = max(*0*, maxprofit(*j* − 1)+p(j)-p(j-1)) = maxprofit(*j*).

(I) <u>Base Case</u>
When there are only two days, there is only 1 transaction possible and let there by 'm' stocks present. We store all our required values in global variables, which will be updated only when the profit of current stock is greater than the maximum profit of all stocks previously computed.
For a single stock, for day 'n ' we calculate the difference between value of stock on day n and day n-1. If the difference is negative means, there being no profit available, and we set profit to 0. Else, we set maxtotal the price difference of these stocks. This is what is what profit will be. For each subsequent stocks, we only update the values if new values are available or else the value remains maximum of profit calculated on previous stocks.
OPT[n]=A[n]-A[n-1]; n=1
=max {OPT(j-1), OPT(j-1) +A[j]-A[j-1]}; otherwise

(II) For nth stock:
Let us assume that this is true for a value n.
Case 1: New profit is less than current maximum profit
If profit[n]<maxtotal, we do not change value of maxTotal i.e., we will choose OPT(j-1). Therefore, at n+1, we have maximum profit as our result by induction.

Case 2: New profit is greater than current maximum profit
If profit[n]>maxtotal we change maxprofit to our current profit and update our maxtotal variable with the new value as OPT(j-1) +A[j]-A[j-1]. Latest maximum profit is chosen by our algorithm. Here, OPT(j-1) is already correct and an additional step to compute current profit will give us our new profit.

Case 3: New profit is negative (subset of case 1)
We keep the latest profit value to be 0 incase the profit is negative. Therefore, it does not affect any values calculated by OPT(j-1) in any way.

Therefore, by induction, we can say that for any optimal OPT(n), our algorithm calculates OPT(n+1) correctly.

Time complexity:

As there is an outer loop i running from 0 to m-1 for each stock inside the function MaxProfitRecMem, and the time spent in single call to maxprofit is O(1),excluding the time spent in recursive calls it generates. So the running time is bounded by a constant times the number of calls issued to maxprofit. Since the implementation itself gives no explicit upper bound on this number of calls, we try to find a bound by looking for a good measure of progress.

The most useful progress measure here is the number of entries in profit that are not “empty.” Initially this number is 0; but each time the procedure invokes the recurrence, issuing one recursive call to maxprofit, it fills in a new entry, and hence increases the number of filled-in entries by 1. Since profit has only n + 1 entries, it follows that there can be at most O(n) calls to maxprofit function, and therefore the running time of maxprofit(n) is O(n). So, the total running time is O(m*n).

Space Complexity:

Since, we are using an extra array profit of the size of n+1 to store the maximum values of the profit, the space complexity is O(n).

3.b) Design of Iterative BottomUp implementation of Alg3:

- We define the function named maxProfitBottUp with the formal parameters passed as the 2d array depicting the prices of the m stocks for n days, m depicting the number of stocks and n as the number of days.
- We initially initialize a variable maxtotal and maxprofit (representing the maximum so far among the prices and the final maximum profit outside the loop for stocks respectively) as 0.
- The variables buyday, sellday, bd, sd, stock are defined to represent the buy day inside n loop, sell day inside n loop, final buy day inside m loop, final sell day inside m loop, and the stock number.
- We run a loop for the number of stocks.
- Let Oj be the maximum possible return by selling the stock on day i(1,2,...,n). O1=0 as the stock cannot be sold on day 1.
- There can be 2 cases for the optimal way of selling the stocks on day j:
  (i) If we are not holding the stock on day i-1, then Oi=0
  (ii) If we are holding the stock on day i-1, then Oj=Oj-1+p(i)-p(i-1)
- Check is carried out for the maxtotal and it is updated to the new maximum profit.

- The sell day is updated using the above check.
- Eventually, the overall maxprofit is calculated and returned and the buy and sell days are printed.

Pseudocode:

```
maxProfitBottUp(int p[][], int m, int n)
For j=0,1,..,m-1
  Maintain maxtotal initialized to 0
  Maintain buyday initialized to 1
  Maintain sellday initialized to 1
  For i=1,2,...,n-1
     profit[i]=maximum(0,profit[i-1]+p[j][i]-p[j][i-1])
     If p[j][i]-p[j][i-1]<0 AND profit[i]=0 then
       buyday=j+1
     Endif
     If mactotal<profit[i] then
       sellday=i+1
     Endif
     maxtotal= Maximum(maxtotal,profit[i])
     Check for maximum between maxtotal and maxprofit and update
   Endfor
   End
```

Recursive Formulation:

Let Xj be the maximum possible return by selling the stock on day j, then:
Xj=max(0,Xj-1+(p(j)-p(j-1)))

Proof of Correctness:

We will prove the correctness using proof by induction. It follows the same approach as given in task 3a.

(i) Base Case
When i is the first day, the profit will be 0. Therefore, we initialize the forest index of profit to 0.

(ii) For nth stock:
Let us assume that this is true for a value n.
Case 1: New profit is less than current maximum profit
If profit[n]<maxtotal, we do not change value of maxTotal i.e., we will choose OPT(j-1). Therefore, at n+1, we have maximum profit as our result by induction.

Case 2: New profit is greater than current maximum profit
If profit[n]>maxtotal we change maxprofit to our current profit and update our maxtotal variable with the new value as OPT(j-1) +A[j]-A[j-1]. Latest maximum profit is chosen by our algorithm. Here, OPT(j-1) is already correct and an additional step to compute current profit will give us our new profit.

Case 3: New profit is negative (subset of case 1)
We keep the latest profit value to be 0 in case the profit is negative. Therefore, it does not affect any values calculated by OPT(j-1) in any way. Since, we are using the state implementation of the form Xj=max(0,Xj-1+(p(j)-p(j-1))), this is taking care of keeping the maximum profit as 0 when it becomes negative.
Therefore, by induction, we can say that for any optimal OPT(n), our algorithm calculates OPT(n+1) correctly.

Time Complexity:

As there are 2 loops: the outer one j running m times and the inner one i running n times, therefore the total running time is O(m*n).

Space Complexity:

Since, we are using an extra array profit of the size of n to store the values of the profit for each day, the space complexity is O(n).

Problem Statement 2:

Given a matrix A of m × n integers (non-negative) representing the predicted prices of m stocks for n days and an integer k(postive), find a sequence of atmost k transactions (buy and sell) that gives maximum profit.

<u>Alg4</u>

Design a Θ(m∗n^2k) time brute force algorithm for solving Problem statement 2:
We set both final op[] ([max profit, stocknumber1, buy1, sell1, stocknumber2, buy2, sell2]) and final profit to 0 and initialize final op[] with zeros.
We pick a pair of buyday-selldays for every stock m in the range of (0 to M) and then figure out the profit for that pair. The algorithm then divides the issue into three parts; the first portion uses a matrix for stocks numbered 1 to m-1 with days from sellday to N.
We take the matrices from sellday to N and m+1 to M for the second scenario.
We use the same stock, m, and an array from sellday+1 to N days for the third scenario.
In order to increase the profit of the initial pair, we then calculate the maximum profit in each of the three scenarios, designated profit2, and add it. Both this value and all other necessary values are stored in final op along with final profit.

Time complexity:

We are using an outer m loop, and four inner loops going upto n days, therefore if we take the base case of k=2, our total running time is O(m*n^2k).

<u>Alg5</u>

Design a Θ(m∗n^2*k) time dynamic programming algorithm for solving Problem statement 2:

- Let T[i][j] represent the array to be filled in the bottom up approach with i transactions and j days.
- We can have 2 cases for selling the stocks on jth day:
  (i) Not transacting on jth day, T[i][j-1]
  (ii) If transacting on jth day ,supposing we bought it on a day m before j,
  T[i-1][m]+prices[j]-prices[m] as for the remaining i-1 transactions removing the ith transaction.
- We run an outer loop x for k transactions, an inner loop y for the n days and a loop inside y as z for m in stocks.
- We initially initialize a variable maxVal (representing the maximum so far) for each stock as 0.
- For each stock, an innermost loop zz is run for the days less than the yth day for the transactions less than the yth day where we can buy the stock to sell at yth day.

- We do this because we need to find the maximum value for each day for each stock and then compare and find the maximum among all.
- Check is carried out for the maxVal and it is updated to the new maximum profit according to the cases mentioned above.
- We initialize the value in the array T[][] by taking the maximum between the maxVal and the previous days' maximum profit so far.
- The value at the last index in row and column , i.e. T[k][prices.length-1] is returned as the maximum profit of the whole transactions for m stocks.
- For calculating the buy and sell days, we retrace the steps back using the function printActualSolution from the maximum profit index in the 2d array.
- We check if the profit in the jth column matches with the profit in the j-1 th column. If it doesn't, then we calculate the difference between the first element in that column with the maximum profit in the last row of the same column.
- We check for this difference in the array from the i-1th row(i.e. the row above, the second last row).
- When we find the difference, that column becomes our buy day. Similarly, we check for k transactions using this approach and output the buy and sell days and the stock numbers.

Pseudocode:

```
maxProfitSlowSolutionM(int prices[][], int K, int m, int n)
If k is 0 OR m is 0 OR n is less than 1 then
  Return max profit as 0
For x=1,2,..,k
  For y=1,2,...,n-1
    For z=0,1,...,m-1
     Maintain a variable maxVal for maximum profit initialized to
     0
      For zz=0,1,...,y-1
        maxVal= Maximum(maxVal, prices[z][y] - prices[z][zz] + T[x-
1][zz])
      Endfor
    T[x][y] = Maximum(Maximum(T[x][y-1], maxVal),T[x][y]);
   Endfor
  Endfor
Endfor
Return T[K][n - 1]
End

printActualSolution(int T[][], int prices[][])
```

Recursive Formulation:

For buyday and cooldown day, the maximum profit is the previously obtained profit, M[i]=M[i-1] –(1)

For sellday j,
We have the buyday at j-c-1 and j-1 as cooldown day, therefore
M[i] = Math.max(M[j -c-1] + profit[i] - profit[j])
Using (1) and (2), we get:-
OPT[i][j]=max(OPT[i][j-1],prices[j]-prices[m]+OPT[i-1][m]) for m=0,1,2,..,j-1 and j=0,1,2,...,n-1

Proof of Correctness:

We will prove the correctness using proof by induction.
(i)Case when K =0 OR m=0 OR n <= 1
In this case, the maximum profit should be 0 as there is no transaction possible. Our algorithm does the same by returning 0 in this case.

(ii) Let us assume this is true for k>0 transactions and number of stocks m>0 by inductive hypothesis.
Case 1: When there is no transaction on day j, our current day, the optimal solution should use the transaction results from the days less than j, i.e. till j-1. Our algorithm does the same giving the maximum profit till j-1.
Case 2: When it is a transaction day, we get the profit by considering any day m prior to j and calculating the maximum profit at j using the recursive formula mentioned above.

Therefore, by induction, we can say that for any optimal OPT(n), our algorithm calculates OPT(n+1) correctly.

Time Complexity:

In total there are 4 loops:

Outermost loop x running k times, innerloop y running n times, second inner z running m times and the innermost zz running till y, so total n times. Therefore the total running time is O(k*n*m*n).i.e. O(m*n^2*k).

Space Complexity:

Since, we are using an extra 2d array of size k*n to store the values of the profit for each day, the space complexity is O(k*n).

Task 6b:

We are given a mxn matrix with predicted prices of m stocks for n days and an integer k as the maximum number of transactions permitted. We are going to design our algorithm with Θ (m ∗ n ∗ k).

Bellman Equation:

OPT[k][j] = max {OPT[1][j-1], OPT[j-1] +A[j]-A[j-1]}; k=1

= max {OPT[k-1] [j-1], max {OPT[k][j], A[j]-A[j-1] +OPT[k][j]}}

Algorithm:

This is a two-part algorithm. In the first part, we fill an additional array of size[k+1] [n] with maximum profits available at each value of k. Now, the value of any k+1 will have a value of k inside it meaning a smaller subproblem. In the second part, we backtrack the solution from the last element of the array to the first element of the array.

Pseudocode:

```
Initialize temporary
For k=1,2,3...., k
Initialize hashmap to maintain maximum difference encountered for k
for j=0,1, 2,…,n
for a=0,1, 2,…,m
tempArray[i][j] = Math.max(Math.max(tempArray[i][j-1],stockPrices[a][j]
    + maxDiff.get(a)),tempArray[i][j]);
                    maxDiff.put(a,Math.max(maxDiff.get(a), tempArray[i-1][j] -
                    stockPrices[a][j]));
            End for
        End For
End For
```

```
Initialize Stacks to PUSH buydate,selldate and stock number
Initialze i, j to last elment of temporary stack
Loop while true
If either I or j reaches 0, break the outer loop
If two elements in same row is same, decrease value of j
Else
Push the latest index of j on stack. It will be our sell date
For k=j-1,j-2,…,0
For a=0,1,2,…,m
Compute maximum difference
if difference of maximum value of previous transaction limit(i) and stock price of any
        stock on jth day are equal to maximum difference
decrease transaction value(i)
make j equal to latest k value
push current j value as buy date
push current a value as stock number
break loop of a
End for
If loop of a is broken by break, then break this loop also
End for
```

Time and Space Complexities:

Time Complexity= Θ $(m * n * k)$

Space Complexity= Θ (k*n)

Proof of Correctness:

Base Case: k=1
When k=1, this algorithm will perform like the algorithm in which there are m stocks and there is only 1 transaction. From Task3b, it gives correct results.
For kth transaction:Let us assume that our algorithm works correctly for kth transaction. For (k+1) we compare the newly computed profit with the profit already present in the

same position for transaction 'k', which is correct(our assumption) . If our newly computed profit is greater than the profit already present for the previous transaction, we have the new value updated in the temporary array which is based on values in the temporary array for kth transaction. In backtracking, we will consider only these latest values when all the temporary array is filled. Therefore, by induction, we can prove that our algorithm is correct.

OPT[k]=max {OPT[k-1], max {OPT[k], A[j]-A[j-1] +OPT[k]}}

Problem Statement 3:

Given a matrix A of m × n integers (non-negative) representing the predicted prices of m stocks for n days and an integer c(postive), find the maximum profit with no restriction on number of transactions. You cannot buy any stock for c days after selling a stock.

Alg8

Design a Θ(m∗n^2) time dynamic programming algorithm for solving Problem3:

- We define the function named maxProfit with the formal parameters passed as the 2d array depicting the prices of the m stocks for n days, m depicting the number of stocks, n as the number of days and c for the cooldown period.
- This is the linear scan backwards technique.
- We'll pay attention to the current day i. 3 cases:
- (i)Purchase Day, (ii) Cooldown Day, (iii) Sale Day
- If it is a buy day, then we are unable to make a profit on the current day; therefore, the maximum profit is the maximum profit we made in the past, or yesterday, M[i] = M[i - 1].
- Similar to what was stated earlier, if it is a cooldown day, we cannot make money today. M[i] = M[i - 1]
- If i were a sell day, the purchasing day may be any day prior to day i. (exclusive, i.e. 0 to i- 1). Let's write it down as j, and then the profit is equals yesterday's profit + profit[i] - profit[j].
- j-1 cannot be a buy day. The stock must be sold before you may buy again as needed because you cannot have 2 consecutive days (j - 1 and j) as the buy day.
- j-1 is not a sell day as well, because then day j must be a cooling day. This is not possible.
- Thus, day j - 1 must be a cooldown day. If it is a cooldown day, then the max profit we can get at day j - 1 is M[j - 2]
- An outer loop i runs for n days and an inner loop j runs for the m stocks calculating the maximum profit till the ith day for each stock.
- An array M[] is defined to store the maximum values of the profits upto day i.

- The base case of the first day is used i.e. on day 1 there can be no sell transaction, therefore M[0]=0.
- For the case of the second day i, we store the difference of price[i] and price[i-1].
- For i greater than 1, first initialize M[i] with the maximum of the the profit so far and the value at M[i].
- A loop k runs for the transactions till i-1,i.e. the current day.
- If k is greater than or equal to the cooldown period, we use a variable prev to store the value of the profit at index i of M so as to satisfy the condition of not buying just after selling a stock because of the cooldown period.
- Store the maximum of the current and the previous added with the difference of consecutive days in M[i].
- M[i] stores the maximum profit encountered so far till the ith index, calculating the difference of each Day starting from 0 to the ith day.
- We return the last index value of M.

Pseudocode:

```
maxProfit(prices[][],c,m,n)
If m is 0 OR n is less than 2 then
  Return 0
Endif
For i=0,1,..,n-1
   For j=0,1,...,m-1
      If i is equal to 0 then
        M[0] is 0
      Endif
      Else
        M[i]=Maximum(M[i-1],M[i])
        For k=0,1,...,i-1
          prev=M[k-c-1] if k>=c+1 OR prev=0 if k<=c+1
         M[i]=Maximum(M[i], prev+prices[j][i]-prices[j][k])
       Endfor
     Endelse
  Endfor
Endfor
Return M[n-1]
End
```

Recursive Formulation:

M[i] = max(M[i - 1], Math.max(M[j - c-1] + nums[i] - nums[j]) , for j = 0 to i - 1

Proof of Correctness:

(i) Base case:(i=0)
When i is zero , the profit will be 0 as there is no buy or sell transaction, therefore this is true for i=0 case

(ii) When i=1:
Using inductive hypothesis, for i=1, i.e. for second day transaction, we will calculate the maximum profit by computing the difference between the previous day if there is a transaction and taking the maximum of the two

(iii) Same holds for the transactions for i>1, by inductive hypothesis.
The maximum profit calculation should take into account the cooldown period and accordingly the buy day for optimal solution. Our algorithm does the same ,therefore preserving the optimality of the solution.

Therefore, for every optimal solution Opt[n], our algorithm returns an optimal solution for Opt[n+1].

Time Complexity:

In total there are 3 loops:
Outermost loop i running n times, innerloop j running m times, and the innermost k running till i-1, so total n times. Therefore the total running time is O(n*m*n).i.e. O(m*n^2).

Space Complexity:

Since, we are using an extra single dimensional array M of size n to store the values of the profit for each day, the space complexity is O(n).

Alg9

Design a Θ(m∗n) time dynamic programming algorithm for solving Problem3:

9.b)

Iterative BottomUp implementation of Alg9:

- To represent the decision at index i:

  (i)buy[i]: Max profit till index i. The series of transaction is ending with a buy.

  (ii)sell[i]: Max profit till index i. The series of transaction is ending with a sell.

- Till index i, the buy / sell action must happen and must be the last action.
  It may not happen at index i. It may happen at i - 1, i - 2, ... 0.
- In the end n - 1, return sell[n - 1]. Apparently we cannot finally end up with a buy.
  In that case, we would take a rest at n - 1.
- For special case no transaction at all, classify it as sell[i], so that in the end, we can still return sell[n - 1].
- buy[i]: To make a decision whether to buy at i, we either take a rest, by just using the old decision at i - 1, or sell at/before i - 2, then buy at i, We cannot sell at i - 1, then buy at i, because of cooldown.
- sell[i]: To make a decision whether to sell at i, we either take a rest, by just using the old decision at i - 1, or buy at/before i - 1, then sell at i.
- The same approach is used as the previous algorithm 8, except that this is the optimized solutio
- We use a variable maxDiff to keep track of the changing difference M[j-c-1]-prices[j].
- Return the last index value of M as maximum profit.

Pseudocode:

```
maxProfit(int prices[][],int c,int m,int n)
Maintain an array M of size n
Maintain an array maxDiff of size m
For i=0,1,..,n-1
  For j=0,1,..,m-1
    //same as above for algorithm 8 for the cases i=0 and i=1
  Else// for i>1
    M[i] = Math.max(M[i - 1], maxDiff + prices[i]);
    maxDiff = Math.max(maxDiff, M[i - c-1] - prices[i]);
  Endelse
 Endfor
Endfor
Return M[n-1]
End
```

Proof of Correctness:

(i) Base case:(i=0)
When i is zero , the profit will be 0 as there is no buy or sell transaction, therefore this is true for i=0 case

(ii) When i=1:
Using inductive hypothesis, for i=1, i.e. for second day transaction, we will calculate the maximum profit by computing the difference between the previous day if there is a transaction and taking the maximum of the two

(iii) For i<2:
There is no possibility of taking into account the maximum profit of M[j-c-1] transactions, therefore it should yield the maximum among the maxDiff and -prices[i] which our algorithm is doing correctly.

(iii) Same holds for the transactions for i>1, by inductive hypothesis.
The maximum profit calculation should take into account the cooldown period and accordingly the buy day for optimal solution. Our algorithm does the same ,therefore preserving the optimality of the solution.

Therefore, for every optimal solution Opt[n], our algorithm returns an optimal solution for Opt[n+1].

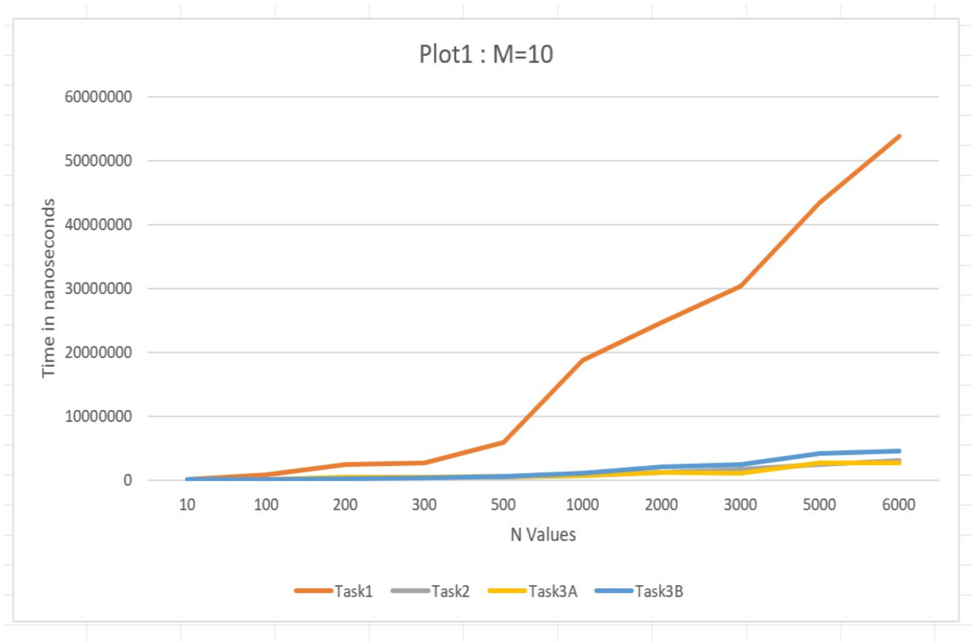
Time Complexity:

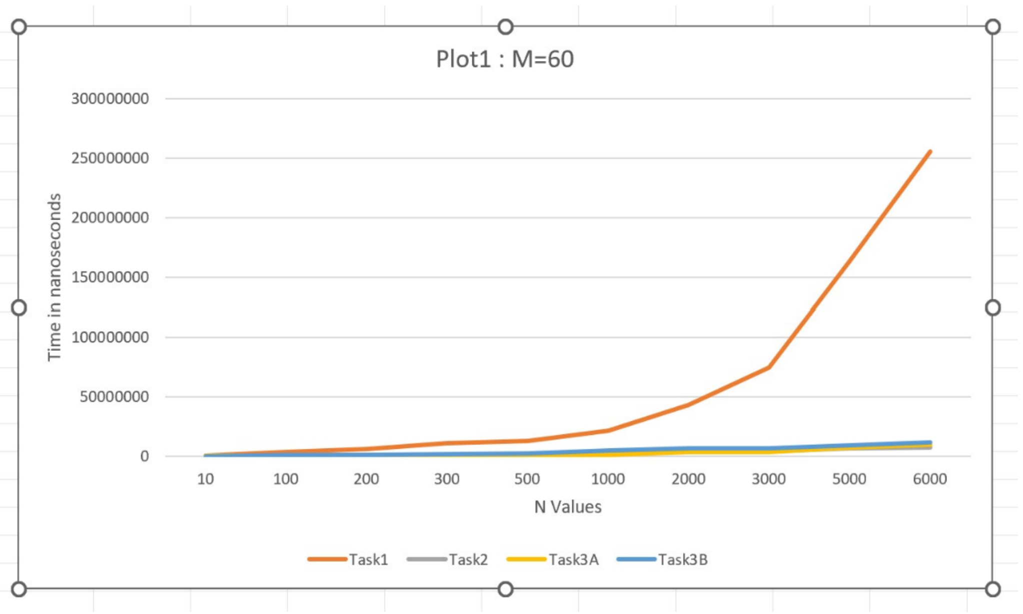
In total there are 2 loops:
Outermost loop i running n times, innerloop j running m times for the stocks. Therefore the total running time is O(m*n).

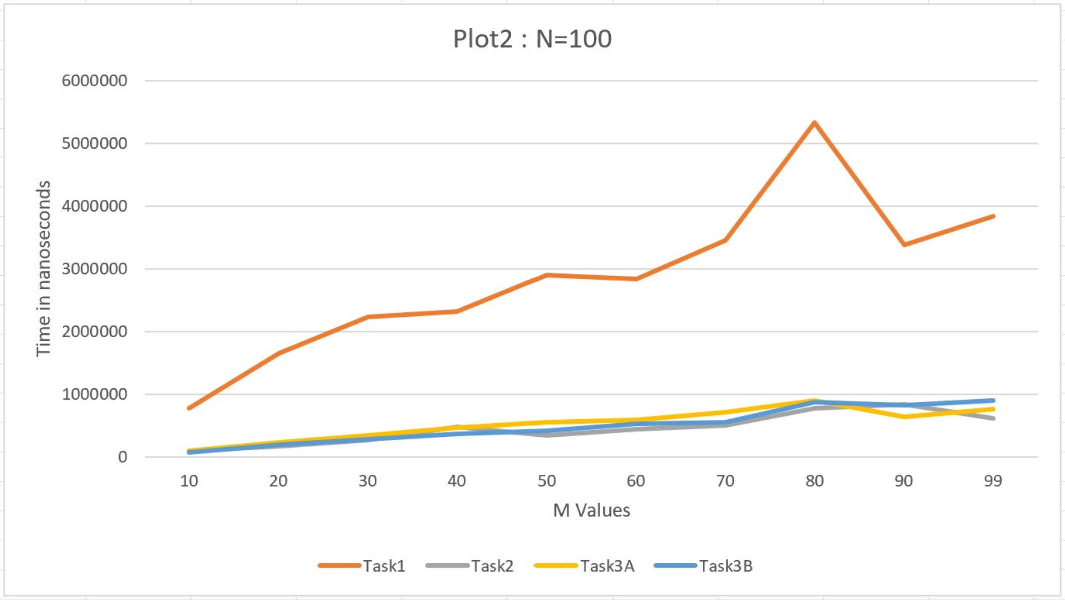
Space Complexity:

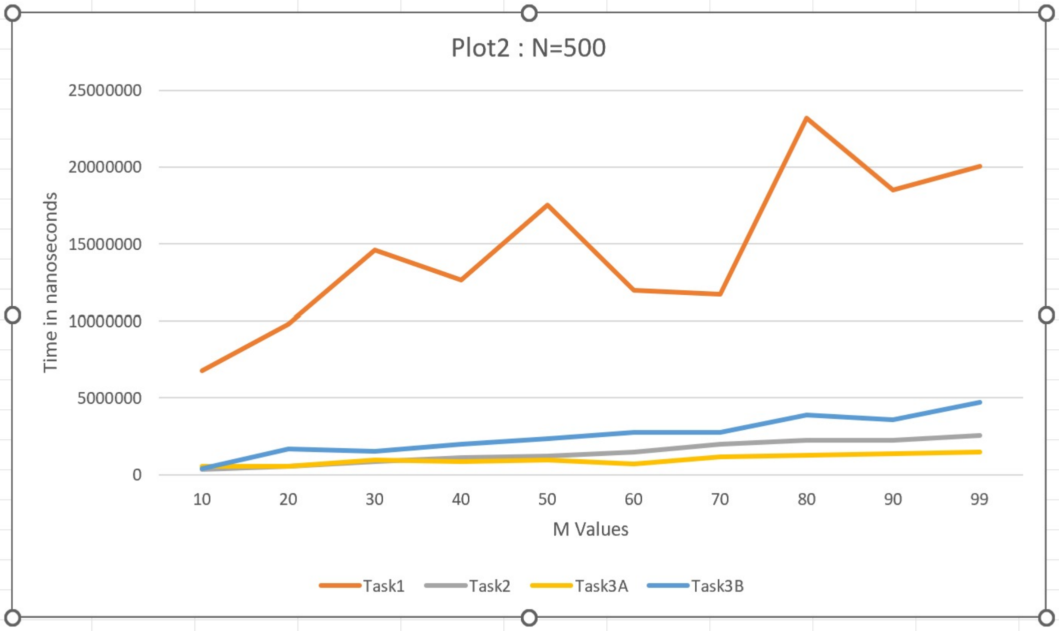
Since, we are using an extra single dimensional array M of size n to store the values of the profit for each day, the space complexity is O(n).

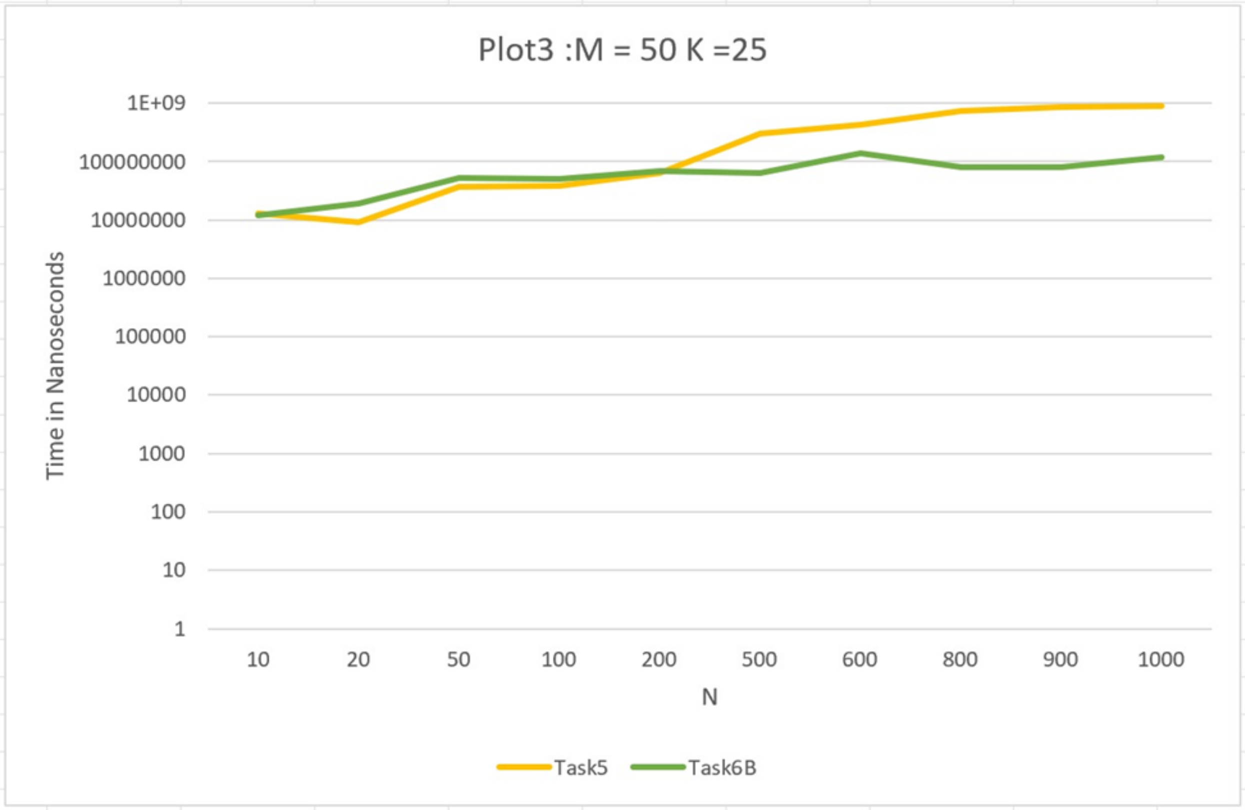
Experimental Comparative Study

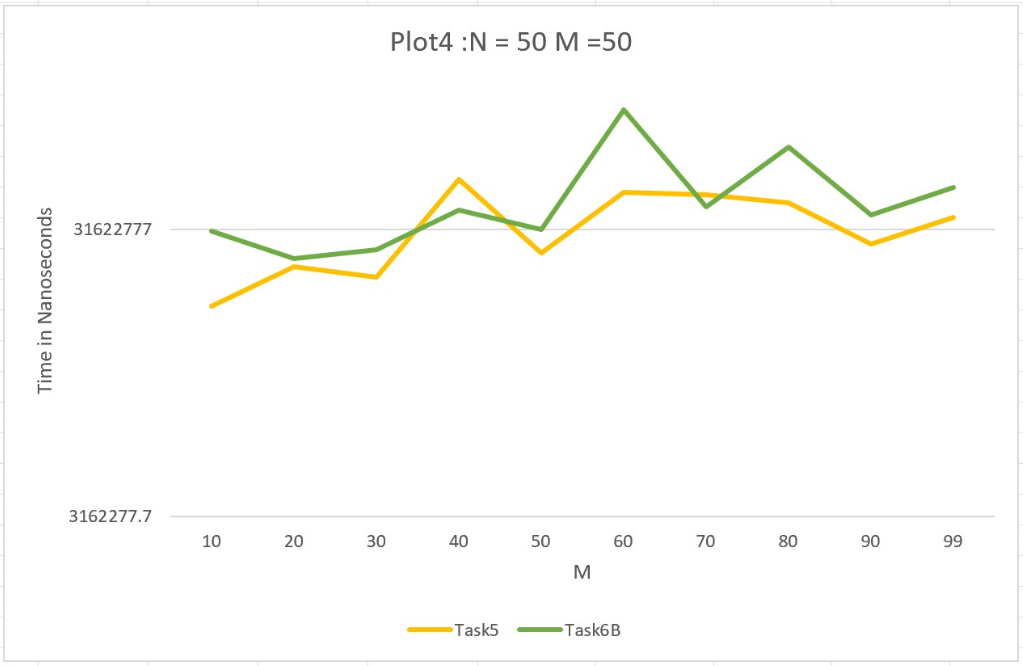


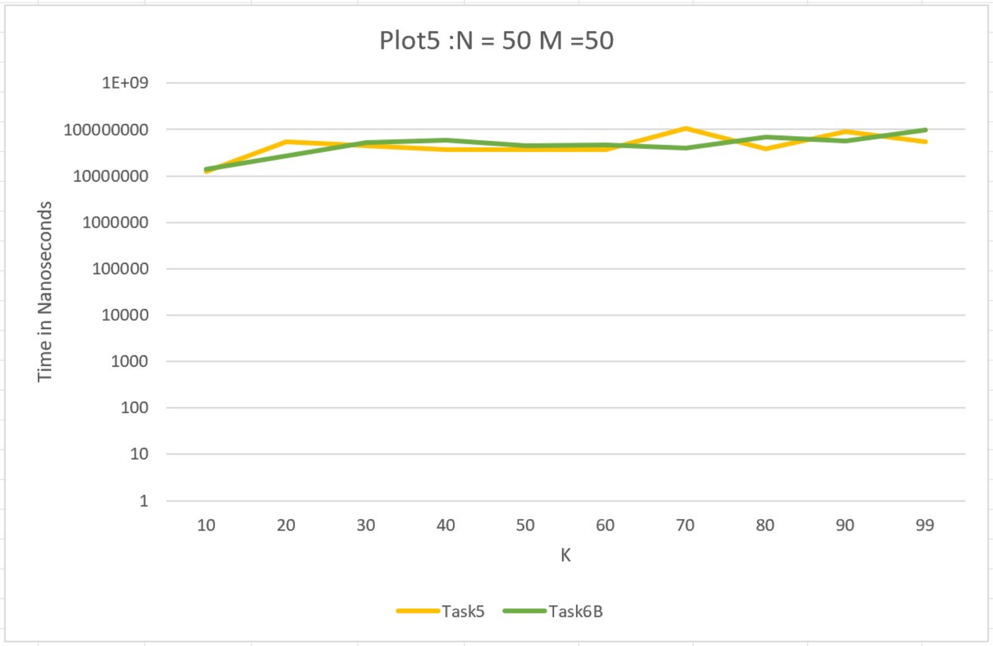


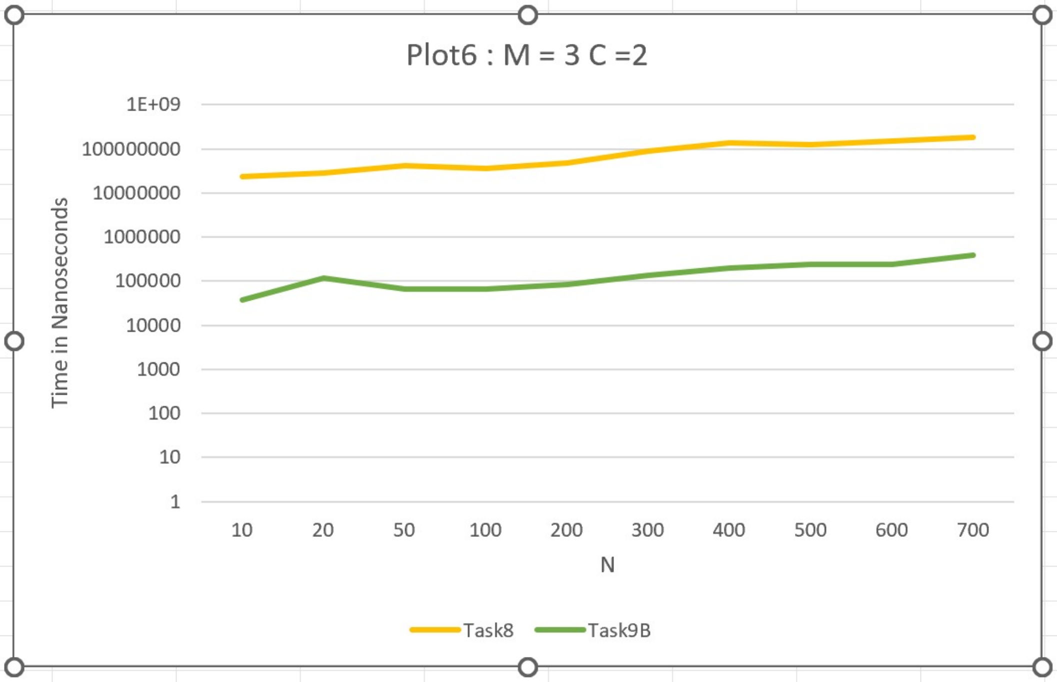

Plot2 : N=100
Time in nanoseconds
6000000
5000000
4000000
3000000
2000000
1000000
0
10
20
30
40
50
60
70
80
90
99
M Values
Task1
Task2
Task3A
Task3B

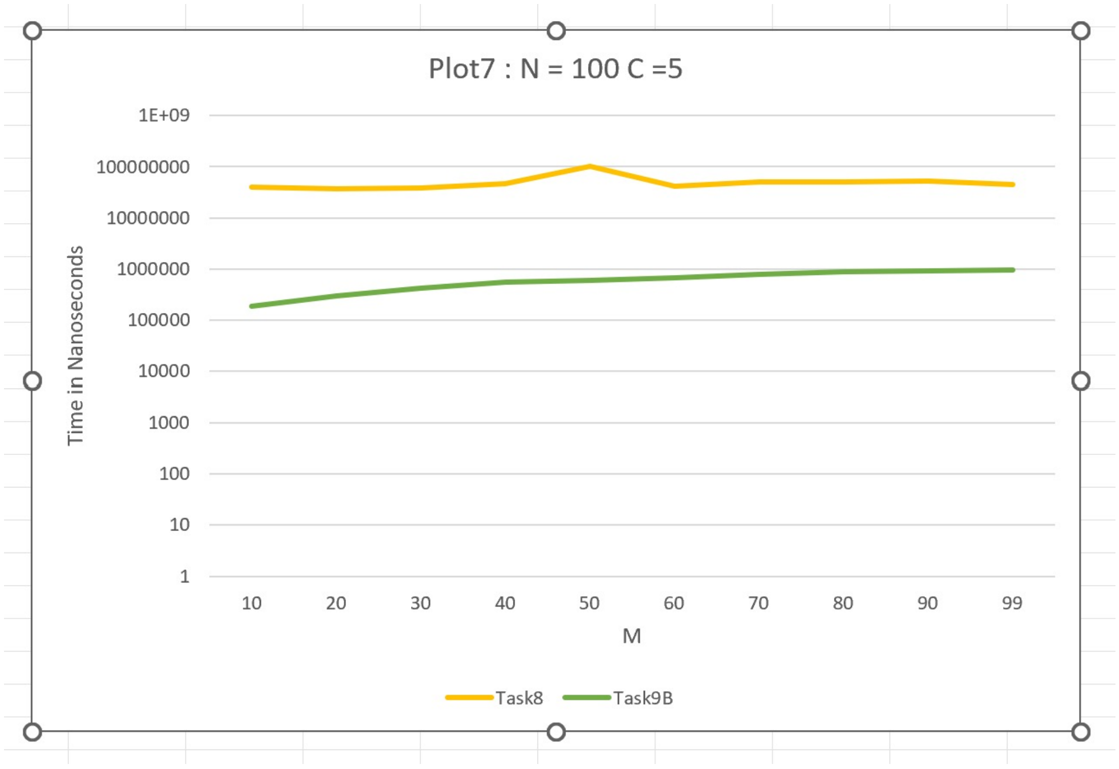

Plot2 : N=500
Time in nanoseconds
25000000
20000000
15000000
10000000
5000000
0
10
20
30
40
50
60
70
80
90
99
M Values
Task1
Task2
Task3A
Task3B


Plot3 :M = 50 K =25
1E+09
100000000
10000000
1000000
100000
10000
1000
100
10
1
Time in Nanoseconds
10
20
50
100
200
500
600
800
900
1000
N
Task5
Task6B


Plot4 :N = 50 M =50
31622777
3162277.7
Time in Nanoseconds
10
20
30
40
50
60
70
80
90
99
M
Task5
Task6B


Plot5 :N = 50 M =50
Time in Nanoseconds
1E+09
100000000
10000000
1000000
100000
10000
1000
100
10
1
10
20
30
40
50
60
70
80
90
99
K
Task5
Task6B


Plot6 : M = 3 C =2
Time in Nanoseconds
1E+09
100000000
10000000
1000000
100000
10000
1000
100
10
1
10
20
50
100
200
300
400
500
600
700
N
Task8
Task9B


Plot7 : N = 100 C =5
1E+09
100000000
10000000
1000000
100000
10000
1000
100
10
1
Time in Nanoseconds
10
20
30
40
50
60
70
80
90
99
M
Task8
Task9B

Conclusion

In each of the programming tasks, we experienced a varying level of difficulty and complexity. Keeping our algorithms tightly bound to the time complexities was a challenge in itself. During the implementation of the programming project, we made a few observations. Firstly, as we move towards tasks which are complex to implement, we observe the running times of the algorithms decrease as is evident from the attached graphs. That is, ease of implementation is inversely proportional to the efficiency of the algorithm. In addition, we observed that in cases where the input size (n) is high, we observed that the recursive algorithms with memoization fared better than all other algorithms. One reason might be because as the value of n increases, there are more overlapping subproblems, which might have given us the result that we observed. In addition, in some cases where the values of m and n are very high, we experience that recursive algorithms with memoization run out of memory in some cases.