COT 5405 -Analysis of Algorithms Project

Dharmam Savani UFID- 9524 9907

Dev Patel UFID- 2328 5413

Chintan Acharya UFID- 6435 4143

UNIVERSITY OF FLORIDA

November 21, 2022

Problem Definition:

We are given an array of price predictions for m stocks for n consecutive days. The price of stock i for day j is A[i][j] for i = 1,...,m and j = 1,...,n. You are tasked with finding the maximum possible profit by buying and selling stocks. The predicted price on any day will always be a non-negative integer. You can hold only one share of one stock at a time. You are allowed to buy a stock on the same day you sell another stock. More formally,

Problem 1: 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.

Problem 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 (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.]

Design and Analysis of Algorithms:

Algorithm1:

Design a Θ(m ∗ n2) time brute force algorithm for solving Problem 1.

Solution:

For the brute force algorithm, the approach is the iterate over all the stocks and find the maximum approach taking into account all the possible buy and sell permutations. The pair for each stock j = 1, …, m will be (a, b) where a = 1, … , n - 1 and b = a + 1, … , n.

Proof of correctness:

This algorithm checks all the possible pairs for each stock and finds the maximum out of all those pairs. Hence, this algorithm will always find the correct desired output and find the maximum possible profit always. Brute force algorithms always find the correct solution.

Time complexity analysis:

The time complexity for the given task can be calculated as follows: First iterating over all the stocks i.e. m. Then all the pairs for that stock in n days will be from a = 1 to n-1 and from b = a+1 to n. So, for each stock on day 1, there will be n - 1 pair for day 2 there will be n - 2 pairs, and so on. Hence there will be (n-1) + (n-2) + … + 1 pairs for each stock i.e. (n-1)(n) / 2 pairs. Hence there will be m * (n(n-1)/2) pairs.

So the time complexity for this algorithm will be $\mathbf{\Theta (m * n^2)}$

Space Complexity analysis:

The algorithm stores the stocks and their prices in a 2d array of size m*n. It also stores the least price, profit, buy and sell as an integer.

So the space complexity for this algorithm will be: **Θ (m * n)**

Algorithm2:

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

Solution:

For the greedy algorithm, the approach is to iterate over all the stocks and for each stock, finding the minimum price of the stock and subtracting it to all the prices of the next days. Thus we find the transaction with the maximum profit for each stock and find out the maximum profit transaction in the process.

So for m stocks for n days, for each stock 1 to m, the algorithm goes from j = 1 to n, keeps a record of the minimum price from 1 to j, and also keeps a record of the maximum profit from days 1 to j. Since a stock can be sold only after the day its bought, the minimum price must be subtracted from the prices after that day only. So, the algorithm greedily keeps a record of the maximum possible profit while going over each day for every stock.

Proof of correctness:

Invariant: MaxProfit(j): The given algorithm will always give maximum profit after j days for n stocks.

Pf. : MaxProfit(j) finds the maximum profit of each stock greedily. Finding min and max price values where the index of min is less than the index of max. Then finds the maximum from all the maximum transactions of each stock. So it will always return the maximum value of the profit.

Time complexity analysis:

The time complexity for the given task can be calculated as follows: First iterating over all the stocks i.e. m. Then the algorithm goes from j = 1 to n and finds the minimum price and in turn finds the maximum possible profit from that stock. So, it finds the maximum profit from all the stocks and finds the maximum out of those transactions giving out the maximum profit and the transaction.

So the time complexity for this algorithm will be **Θ (m * n)**

Space Complexity analysis:

The algorithm stores the stocks and their prices in a 2d array of size m*n. It also stores the least price, profit, buy and sell as an integer.

So the space complexity for this algorithm will be: **Θ (m * n)** [Here c is a constant]

Algorithm3:

Design a $\Theta(m * n)$ time dynamic programming algorithm for solving Problem 1.

Solution:

For the dynamic programming, for each stock, it checks if the minimum price yet is greater than the current price of the stock then it sets the minimum price as the current price. Then it calculates the profit by selling the stock at the current price after buying it at the minimum price. It checks if the current profit is greater than the previous profit then it sets the max profit to the current profit and calls the next iteration with the new minimum price or new max profit. So it finds the maximum profit for each stock and outputs the maximum out of those.

Def: OPT(j) - Maximum profit for a stock for days 1 to j.

Goal: Max{ OPT_i (n) } where i = 1 to m

Case 1: OPT(j) sells the stock on day j

- Current profit will be the price of the stock on day j minus the buying price

- If the current profit is greater than the max profit so far, set the max profit to the current profit

Case 2: OPT(j) buys the stock on day j

- Set buying price as the price on the stock on day j

Case 3: OPT(j) does not buy or sell the stock on day j

- Current profit must be less than the max profit made so far.

Bellman Equation:

OPT(j) = 0 if j = 0

= max(max(current profit, max profit so far), OPT(j - 1)) if j > 0

Proof of correctness:

Invariant: OPT(j) gives maximum profit from the stocks after j days.

Pf. : Proof by induction

Base Case: OPT(0), OPT(1) return 0

Next Case: OPT(2) = max{OPT(1) which is 0 and since n is 2 there are just 2 days for all stocks and hence just the maximum difference between the second day and the first day for all stock will be the maximum profit}

Inductive Hypothesis: Assuming OPT(j) gives maximum profit.

For OPT(j + 1):

- If the price of the stock is less than the current buying price it will be updated.
- If the current price - buying price is less than the max profit so far it will return the same as the last iteration else it will update the maximum profit and selling date.

Hence for any stock on any day j, it will give the maximum profit.

Time complexity analysis:

The time complexity for the given task can be calculated as follows: First iterating over all the stocks i.e. m. For each stock, it calculates the new buying price or the maximum profit by doing at most 3 operations and calculates the maximum profit till the current day for each day n.

So the time complexity for this algorithm will be O (m * n)

Space Complexity analysis:

The algorithm stores the stocks and their prices in a 2d array of size m*n. It also stores the least price, profit, buy and sell as an integer.

The memoization approach: It uses 2 extra arrays of size n for each stock for storing the minimum prices till the day and the index of the minimum price. So the space complexity for the algorithm will be **Θ (m * n^3)** [Here c is a constant]

The top-down approach: It doesn't use any extra memory except storing the minimum price so far and the maximum profit so far. So the space complexity for this algorithm will be: **Θ (m * n)**

Algorithm4:

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

Solution:

For this task, we need to design a brute-force algorithm. And as we know in the brute force algorithm we will try all possible pairs and output those transactions that will result in maximum profit. First, we will compute all possible pairs of transactions which for n days will be n * (n-1) / 2 i.e. order of n^2. We will compute n^2 pairs in each stock and we will also keep track of previous stock transactions so that we can filter out the most profitable transactions among the two. By filter, we will see if in each transaction with the same buy and sell date we will keep the most profitable one. Now all that's left is to pick k possible pairs from this n^2 transactions.

Proof of Correctness:

In this algorithm, we try all pairs possible for m stocks, and then we pick only those `almost k` pairs that result in maximum profit and are compatible with each other. Hence, as we tried all pairs and using comparison we kept the most profitable ones, we can say that the algorithm outputs at most k pairs with maximum profit.

Time Complexity analysis:

We can see that Get All transactions will result in a time complexity of the order of n^2 also filtering and keeping the most profitable ones will also have a time complexity of the order of n^2, as we will just iterate and compare those two, and updating previous will result in constant time complexity. Now, this results in the time complexity of $m*n^2$, As we iterate over m stocks. Now, as for the time complexity of GetKtransactions, we know that we are picking at most k pairs from n^2 choices, which means it tries all n^2Ck pairs which reduces its time complexity to n^{2k}, and since we are iterating over m stocks, the time complexity is of the order of $\mathbf{\Theta(m*n^{2k})}$.

Space Complexity analysis:

`transactions` take space in order of $m*n^2$ as we try all possible pairs, Now, PickKtransactions will try up to n^{2k} pairs but since we pass the solution to the function and we have the upper bound of k transaction on the solution it will have space complexity in order of k. Hence this gives us an upper bound of $\Theta(m*n^2)$ space.

Algorithm5:

Design a $\Theta(m * n^2 * k)$ time dynamic programming algorithm for solving Problem2.

Solution:

For dynamic programming,

Def: OPT[k][i] - Maximum profit in i days after k transactions.

Goal: OPT[K][N] -> return K tuples of transactions.

Case 1: OPT[i][j] base case

Case 2: OPT[i][j] Number of transactions is greater than the number of days.

- The number of transactions cannot be greater than days so just return the pro

Case 3: OPT[i][j] Number of transactions is less than the number of days.

- Now we have two choices to pick up from. First, pick up the profit from the previous day i.e no transactions on the current day

- Second, if the transaction occurs on the current day. To find that out we find the price of all the stocks on the current day and check which stock gives the maximum profit if we buy the stock on any of the previous days.

Bellman Equation:

OPT[i][j] = 0 if i = 0, j = 0

= OPT[i - 1][j] if i > j

= Max{ OPT[i][j - 1], Max{ For all stocks from 0 to m (Max { For k from 0 to j-1: OPT[i-1][j-1] - A[i][k] } + A[i][j] } } if i <= j

Proof of correctness:

Invariant: OPT[i][j] gives maximum profit.

Pf: Proof by induction.

Base case: i = 1, j = 1 is easy Return 0

Next case: i = 1, j = 2 has 2 days for all days so just return the max of the difference between the second day and the first day.

Next case: i = 2, j = 2 return Max{ OPT[i][j - 1], Max{ all the stock transactions since there will only be one transaction per stock, excluding the previous one } }

Inductive Hypothesis: Assuming OPT[i][j] gives maximum profit after i transactions.

- If i + 1 > j. The number of transactions cannot be more than the number of days so just return the maximum profit after i transactions. So it will return the maximum profit
- If i <= j:
    - Let profit by selling any one of them on a j^{th} day is greater than the profit of j - 1 day. Then OPT will return that value.
    - If the profit by selling is less, the OPT will return the Opt[i][j - 1] value.

Time complexity analysis:

The time complexity for the given task can be calculated as follows: First going from transactions i = 0 to k, then iterating over j = 0 to n days, for each stock we find the maximum profit from k = 0 to j - 1 day to find out the maximum profit for that day after i transactions.

So the time complexity for this algorithm will be: **$\Theta (m * n^2 * k)$**

Space Complexity analysis:

The algorithm stores the stocks and their prices in a 2d array of size m*n. It also stores the least price, profit, buy and sell as an integer.

Storing the OPT in a 2d array of k + 1 length and width n having a tuple of 3 integers as each element. And the result will be stored in a 2 d array of max length k. And some iterating variables for the loops.

So the time complexity for the implementation will be: **Θ (n * (m + k))**

Algorithm 6:

Design a Θ(m ∗ n ∗ k) time dynamic programming algorithm for solving the Problem

Solution:

For dynamic programming, we intend to find out the maximum profit using k or less than k transactions and using all the stocks.

Let K be the maximum number of transactions possible and N be the total days available.

Def: OPT(k,n) - Maximum profit earned by doing at most k transactions until n days using any stocks.

Goal: Max{ OPT_i (K, N) } where i = 1 to m

Case 1: OPT(k,n) sells the stock on day n

- Current profit will be a maximum of two choices

1. OPT(k , n-1)
2. max(OPT(k-1, n-1) - A[m][n-1] + A[m][n]))

profit on day n-1 and transaction k-1 added with transaction profit

- If the current profit is greater than the max profit so far, set the max profit to the current profit

Case 2: OPT(k,n) does not sell the stock on day n

- Current profit must be equal to max profit made so far i.e. OPT(k,n-1)

Bellman Equation:

OPT[k][n] = {

0, k = 0 or n = 0

Max {

OPT[k][n-1],

Max { Max* { OPT[k-1][n-1] - A[M][n-1] } + A[M][n] } for M = 1 to m

}

}

*** -** We store the difference value i.e max yet, which results in the reduction of time complexity by removing a `for` loop.

Proof of correctness:

Invariant: OPT[k][n] gives maximum profit for at most i transactions and j the day.

Proof: Proof by induction.

Base case: k=1 or n = 1 then the maximum profit is 0.

Next case: k = 2, n = 2 maximum profit is greatest of following values:-

1. OPT(k , n-1)
2. Max (OPT(k - 1 , n - 1) - A [M][n - 1] + A [M][n])) where M = 0 to m.

Inductive Hypothesis: Assuming OPT[i][j] gives maximum profit after i transactions.

If **case 1**: If profit by selling is less, the OPT will return the OPT(k, n-1) value which is the maximum profit earned without the inclusion of day n.

If **case 2**: if profit by selling on day n is more than case 1, that means one of the stocks sold on day n will generate more profit than the previous day.

Thus, this shows that all possibility of generating a maximum profit at day n has been covered in my case 1 and 2. So, for any value of k and n, maximum profit can be achieved by using previously calculated optimal values of OPT(k, n-1) and OPT(k - 1, n - 1).

So, by induction, we can deduce that any value of k and n can obtain its optimal value through this efficient algorithm.

Time complexity analysis:

The time complexity for the given task can be calculated as follows: First iterating over possible values of several transactions i.e. k. For each day, it calculates the maximum profit that is possible to earn while iterating over all stocks i.e. m.

So the time complexity for this algorithm will be: **O (m * n * k)**

Space Complexity analysis:

The algorithm stores the stocks and their prices in a 3d array of size m*k*3. For each transaction and day, it stores 3 integers namely maximum profit earned at that point, stock sold to make maximum profit, and Day at which Stock was bought.

For **memoization**:

So the space complexity for the algorithm will be: O (m * k * 3 + c) = **O (m * k * 3)** [Here c is a constant]

For a **top-down approach**:

So the space complexity for the algorithm will be: O (m * k * 3 + c) = **O (m * k * 3)** [Here c is a constant]

Algorithm 7:

Design a $\Theta(m * 2^n)$ time brute force algorithm for solving Problem3

Solution:

For the brute force algorithm, the approach is the to create a recursive function which tries to compute maximum profit obtained by finding all the possible transactions.

Proof of correctness:

This algorithm checks all the possible transactions possible for each day and finds out the maximum profit possible from all possibilities. Hence, this algorithm will always find the correct desired output. Brute force algorithms always find the correct solution. All the possible transaction on the current day can be found as follow:

- If a stock has already been bought then it can either be sold or retained on that day.
- Otherwise, any of the stock can be bought on a particular day considering no transactions have taken place since the last c days or we can skip buying any stock on that day.

Time complexity analysis:

Time complexity for the given task can be calculated as follows:

First iterating over all days. Then find the maximum profit from all possible states on that day. For each day there are at 2 states possible. So, the time complexity is

N = Total number of days.M = the Total number of Stocks

T(n, m) = {

If Stock has been already bought,	2*T(N-1) + θ(M)
If no stock has been bought,	2*T(N-1)

}

So the time complexity for this algorithm will be: O ($M * 2^N$)

Space Complexity analysis:
The algorithm stores the stocks and their prices in a 2d array of size m*n. It also stores all possible transactions. So the maximum number of possible transactions to store is N/(C+2). Each transaction has 4 integers stored in it. So overall space size comes up to N/(C+2)*4.
So the space complexity for this algorithm will be O (m * n + N/(C+2)*4) = **O (m * n)** [Here C is a cooldown period]

Algorithm 8:

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

Solution:

For dynamic programming, we intend to find out the maximum profit using compatible transactions using all the stocks.

Def: OPT(M, N) - Maximum profit earned by using first m stocks until n days.

Goal: Max profit earning at OPT (m,n)

Where m and n represent total stocks and total days.

Bellman Equation:

OPT(M , N) = {

0 , n = 0;

Max (OPT(M , N-1) , OPT(M - 1 , N) , Max* (A[M][N] - A[M][k] + OPT[m -1][Max(0, k - c - 1)) k = 0 to N-1), Otherwise;

}

Here, m is total number of stocks.

Proof of correctness:

Invariant: OPT(m, n) gives maximum profit for first m stock and n th day.

Proof: Proof by induction.

Base case: n = 1 is the maximum profit is 0.

Next case: k = 2, n = 2 maximum profit is greatest of following values:-

- OPT(m , n-1)
- OPT(m-1 , n)
- Max(A[M][N] - A[M][k] + OPT[m -1][Max(0, k - c - 1))) where M = 0 to m.

Inductive Hypothesis: Assuming OPT[m][n] gives maximum profit after i transactions.

If **case 1**: profit earned at day n-1 is highest. In this case, stock m is not sold on day n.

If **case 2**: profit earned using first m-1 stocks is highest. In this case, stock m is not sold on day n.

If **case 3**: profit earned by selling stock m is highest. In this case, stock m is sold on day n. Also, maximum profit is earned by selling on day k. So, along with profit from this transaction, we can also add profit earned by using all stocks and day k - c.

[c is cooldown period]

Thus, this shows that all possibility of generating a maximum profit at day n has been covered in my case 1, 2 and 3. So, for any value of m and n, maximum profit can be achieved by using previously calculated optimal values from cases 1, 2, and 3.

So, by induction, we can deduce that any value of m and n can obtain its optimal value through this efficient algorithm.

Time complexity analysis:

Time complexity for the given task can be calculated as follows we iterate n to find buy and sell day which has the complexity of (n)*(n-1)/2 and all these operations are done m times because it is done for each stock.
So the time complexity for this algorithm will be O (M * N * (N - 1) / 2) = **O (M * N^2)**

Space Complexity analysis:
The algorithm stores the stocks and their prices in a 2d array of size m*n.
So space complexity of this algorithm is **O (m * n)**

Algorithm 9:

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

Solution:

For dynamic programming, we intend to find out the maximum profit using compatible transactions using all the stocks.

Def: OPT(M, N) - Maximum profit earned by using first m stocks until n days.

Goal: Max profit earning at OPT (m,n)

Where m and n represent total stocks and total days.

Bellman Equation:

OPT(M, N) = {

0, n = 0;

Max (OPT(M, N-1), OPT(M - 1, N), A[M][N] - A[M][k] + OPT[m -1][Max(0, k - c - 1) where k is stored and updated whenever we encounter max profit while buying at day k for m stock), Otherwise;

}

Here, m is the total number of stocks.

Proof of correctness:

Invariant: OPT(m, n) gives maximum profit for first m stock and n th day.

Proof: Proof by induction.

Base case: n = 1 is the maximum profit is 0.

Next case: k = 2, n = 2 maximum profit is greatest of following values:-

- OPT(m , n-1)
- OPT(m-1 , n)
- Max(A[M][N] - A[M][k] + OPT[m -1][Max(0, k - c - 1))) where M = 0 to m.

Inductive Hypothesis: Assuming OPT[m][n] gives maximum profit after i transactions.

If **case 1**: profit earned at day n-1 is highest. In this case, stock m is not sold on day n.

If **case 2**: profit earned using first m-1 stocks is highest. In this case, stock m is not sold on day n.

If **case 3**: profit earned by selling stock m is highest. In this case, stock m is sold on day n. Also, maximum profit is earned by selling on day k. So, along with profit from this transaction, we can also add profit earned by using all stocks and day k - c.

[c is cooldown period]

Thus, this shows that all possibility of generating a maximum profit at day n has been covered in my case 1, 2 and 3. So, for any value of m and n, maximum profit can be achieved by using previously calculated optimal values from cases 1, 2, and 3.

So, by induction, we can deduce that any value of m and n can obtain its optimal value through this efficient algorithm.

Time complexity analysis:

Time complexity for the given task can be calculated as follows. We iterate each day while also iterating on each stock.
So the time complexity for tasks 9a and 9b will be O (m * n) = **O (m * n)**

Space Complexity analysis:

The algorithm stores the stocks and their prices in a 2d array of size m*n. So the space complexity for tasks 9a and 9b will be O (m * n) = **O (m * n)**

Implementation tasks:

Task1:

Give an implementation of Alg1.

Pseudo Code:

Step 1 : Input A[m][n]

Step 2 : profit = 0, stock = 0, buy = -1, sell = -1

Step 3 : For I = 1 to m :

```
        For j = 1 to n - 1:
            For k = j + 1 to n:
            if (A[i][k] - A[i][j] > profit)
                profit = A[i][k] - A[I][j]
                stock = i
                buy = j
                sell = k
```

Step 4: Return {Profit, stock, buy, sell}

Task2:

Give an implementation of Alg2.

Pseudo Code:

Step 1: Input A[m][n]

Step 2: profit = 0, stock = 0, buy = -1, sell = -1, leastPriceSoFar = INT_MAX, minIndex = -1

Step 3: For I = 1 to m :

For j = 1 to n:

if leastPriceSoFar > A[I][j]

leastPriceSoFar = A[i][j]

minIndex = i

if profit < (A[i][j] - leastPriceSoFar)

profit = A[i][j] - leastPriceSoFar

buy = minIndex

sell = i

Step 4: Return {Profit, stock, buy, sell}

Task3a:

Give a recursive implementation of Alg3 using Memoization.

Pseudo Code:

OPT(A[], index, profit, minPrice)

Step1: if A[index] < minPrice

minPrice = A[index]

Step 2: currentProfit = A[index] - minPrice

Step 3: if currentProfit > profit

profit = currentProfit

Step 4: return max{ profit, OPT(A, index - 1, profit, minPrice) }

MAIN()

Step 1 : Input A[m][n]

Step 2 : profit = 0, buy = -1, sell = -1, stock = 0;

Step 3 : For i = 1 to m :

result = OPT(A[i], n, profit, INT_MAX) // *BOTTOM UP*

if result > profit

profit = result

Step 4: Return profit

Task3b:

Give an iterative bottom-up implementation of Alg3.

OPT(A, n)

Step 1: dp[n], buyIndex[n], profit = 0

Step 2: For j = 1 to n :

if A[j] < dp[j - 1]

dp[j] = A[j]

buyIndex[j] = j

else

dp[j] = dp[j - 1]

buyIndex[j] = buyIndex[j - 1]

Step 3: For j = 1 to n :

if profit < (A[j] - dp[j])

profit = A[j] - dp[j]

Step 4: return profit

MAIN()

Step 1 : Input A[m][n]

Step 2 : profit = 0, buy = -1, sell = -1, stock = 0;

Step 3 : For i = 1 to m :

result = OPT(A[i], n)

if result > profit

profit = result

Step 4: Return profit

Task4:

Give an implementation of Alg4**.**

Pseudo Code:

PickKtransactions(transactions, index, Solution = ϕ, k) {

if k is zero or index is the length of transactions then → Return Solution

current → transaction[index];

if the current is Compatible with the Solution then{

Solution1 → PickKtransactions(transactions, index+1, Solution U {current}, k-1)

Solution2 → PickKtransactions(transactions, index+1, Solution, k)

if the profit of Solution1 is greater than Solution2 then → Return Solution1

else → Return Solution2

}

else return PickKtransactions(transactions, index+1, Solution, k)

}

BruteForce(Stocks S, Days D, Max Transactions k)

{

previousTransactions → ϕ

for i → 1 to S:

transaction → Get All transactions(i)

transactions[i] → filter_keepMostProfitableOnes(transaction, previousTransactions)

update previousTransactions

maxprofit → ϕ

result → ϕ

for i → 1 to S:

r → GetKtransactions(transactions[i], 1, ϕ, k)

if profit or r > maxprofit then {

```
                result → r
                maxprofit → profit of r
            }
        return result
}
```

Task5:

Give an implementation of Alg5.

Pseudo Code:

Step 1: Input m,n, A[m][n].

Step 2: Initialize OPT[k+1][n]

Step 3: for i = 1 to k+1

```
        for j = 1 to n
            if( j > i)
                OPT[i][j] = OPT[i-1][j]
                continue to the next iteration
            maxProfit = OPT[I][j-1][profit]
            for p = 0 to m
                diff = INT_MIN, index = 0
                for l = 0 to j
                    currentDiff = OPT[i - 1][l][profit] - A[p][l]
                    if currentDiff > diff
                        diff = currentDiff
                        index = l
            profit = A[p][j] + diff
            if maxProfit < profit
                OPT[i][j][profit] = profit
```

OPT[i][j][stock] = p

OPT[I][j][buy] = buy

// BACTRACKING

Step 5: Initialize i = k + 1, j = n

Step 4: while i != 1 and j != 1

if i == 1

j = j -1

else if j == 1

i = i - 1

else if OPT[i][j][profit] == OPT[I][j-1][profit]

j = j - 1

else

Add the transaction OPT[i][j] to result

Step 6: Return result

Task6a:

Give a recursive implementation of Alg6 using Memoization.

Pseudo Code:

Task6A(A, m, n, k)

Step1: OPT → Ø, DiffSoFar → Ø

// Both OPT and DiffSoFar are 2D arrays where each element is a set of three and two objects respectively

// OPT[i][j] = {profit, Stock, buy}, we will access them as OPT[i][j].profit e**tc.**

// DiffSoFar[i][j] = { Difference, day }, we will access them the same as OPT

Step 2: DPMemoization(A, OPT, DiffSoFar, k, n) // Populate OPT

Step 3: return BackTrack(OPT)

DPMemoization(A, OPT, DiffSoFar, current transaction index T, current day D)

Step1: if OPT[i][j] is not Ø then return OPT[i][j] // As this is recursive we can have multiple calls to the same subproblem

Step 2: if T is 1 or D is 1 then

OPT[T][D] → {0, 0, 0} // initialize base case

Return OPT[T][D]

Step 3: If T > D then

OPT[T][D] = DPMemoization(A, OPT, DiffSoFar, T-1, D)

// If T > D i.e. Transactions greater than days we need to return the previous transaction value which is obvious

else

NoTransactionTodayProfit → DPMemoization(A, OPT, DiffSoFar, T, D-1)

maxProfit → NoTransactionTodayProfit.profit

Stock → NoTransactionTodayProfit.stock

Buy → NoTransactionTodayProfit.buy

IfTransactionTodayProfit → DPMemoization(A,OPT, DiffSoFar, T-1, D-1).profit

For i = 1 to m:

Diff → IfTransactionTodayProfit - A[i][D-1]

If Diff > DiffSoFar[T][i].difference then

DiffSoFar[T][i].difference = Diff

DiffSoFar[T][i].day = D-1

Current profit → A[i][D] + DiffSoFar[T][i].profit

If maxProfit < Current Profit then

maxProfit = Current profit

Stock = i

Buy = DiffSoFar[T][i].day

OPT[T][D] ← { max profit, Stock, Buy }

Step 4: return OPT[T][D]

MAIN()

Step 1: Input k, m, n, A[m][n]

Step 2:

result = Task6A(A[i], m, n, k) // *BOTTOM UP*

Step 3: Return result

Task6b:

Give an iterative bottom-up implementation of Alg6.

Pseudo Code:

Task6B(A, m, n, k)

Step 1: OPT → Ø

Step 2: For i = 1 to k:

DiffSoFar = Φ // Here DiffSoFar is the same as above just as a Single Dimensional array with the same properties as in 6A

For j = 1 to n:

If i > j

OPT[i][j] = OPT[i-1][j]

Continue

maxProfit → OPT[i][j-1].profit

Stock → OPT[i][j-1].stock

Buy → OPT[i][j-1].buy

For p = 1 to m:

Diff → OPT[i-1][j-1].profit- A[p][j-1]

If Diff > DiffSoFar[p].difference then

DiffSoFar[p].difference = Diff

DiffSoFar[p].day = j-1

Current profit → A[p][j] + DiffSoFar[p].profit

If maxProfit < Current Profit then

maxProfit = Current profit

Stock = p

Buy = DiffSoFar[p].day

OPT[T][D] ← { maxProfit, Stock, Buy }

Step 3: return **BACKTRACKERSULT(OPT)**

BACKTRACKERSULT(OPT)

Step 1: i = m-1, j = n-1, result = Φ

Step 2: While(i or j is not 0)

Current = OPT[i][j

If i is zero // base case

j -= 1

else If j is zero //base case

i -= 1

else if OPT[i][j].profit == OPT[i][j-1].profit // previous day

j -= 1

else

// Transaction occurred

result = result U { { current.stock, current.profit, current.buy, j } }

// Go to profit where we buy from

j = buy

i -= 1

Step 3: return result

MAIN()

Step 1: Input k, m, n, A[m][n]

Step 2:

result = Task6A(A[i], m, n, k) // *BOTTOM UP*

result = Task6B(A[i], m, n, k) // *MEMOIZATION*

Step 3: Return result

Task7:

Give an implementation of Alg7.

Pseudo Code:

Step 1 : Input A[m][n], C

Step 2 : profit = 0, stock = 0, buy = -1, sell = -1

Step 3 : Function BruteForce2n(current_day, bought_stock , bought_day, Buy)

Base case: current_day >= N

Return 0

If Buy == True

Transactions_didnt_buy = BruteForce2n(A, current+1, -1, -1, buy, c)

Transactions_did_buy = for i = 1 to m (MAX(BruteForce2n(A, current+1, i, current, false, c)))

Return MAX(Transactions_didnt_buy, Transactions_did_buy)

elseTransactions_didnt_sell= BruteForce2n(A, current+1, S, D, false, c)

Transactions_did_sell = BruteForce2n(A, current+c+1, -1, -1, buy, c)

Transactions_did_sell.add(new transaction {S, A[S][current] - A[S][D], D, current});

return MAX(Transactions_didnt_sell, Transactions_did_sell);

Task8:

Give an implementation of Alg8.

Pseudo Code:

Task8(A, m, n, c)

Step 1: DP, DiffSoFar, Buy → Φ

Step 2:

```
For j = 1 to n:
        For i = 1 to m:
                For k = 1 to j:
                Current = DP[i][j]
                        PreviousDayProfit = DP[i][j-1]
                        previousStockProfit = i == 0 ? 0 : DP[i-1][j]
                        profitIfWeBuy = A[i][j] + DP[m-1][max(0,k-c-1]

                DP[i][j] = max(PreviousDayProfit, previousStockProfit,profitIfWeBuy
)
                        If current != DP[i][j] then
                                Buy[i][j] = k
```

Step 3: return BACKTRACKERSULT(OPT, BUY, A)

BACKTRACKERSULT(OPT, BUY, A)

Step 1: i = m-1, j = n-1, result = Φ

Step 2: While(j is not 0)

```
        Current = OPT[i][j

        If i is zero and OPT[i][j] == OPT[i][j-1]

                j -= 1

        else If i > 0 and OPT[i][j] == OPT[i-1][j]

                i -= 1

        else if OPT[i][j] == OPT[i][j-1]

                j -= 1

        else

                // Transaction occurred

                result = result U { { i, A[i][j] - A[i][Buy[i][j]], Buy[i][j], j} }

                // Go to profit where we buy from

                j = buy[i][j] - c - 1

                i = m - 1
```

Step 3: return result

MAIN()

Step 1 : Input c, m, n, A[m][n]

Step 2 :

result = Task8(A, m, n, c)

Step 3: Return result

Task9A:

Give a recursive implementation of Alg9 using Memoization.

Pseudo Code:

Task9A(A, m, n, c)

Step 1: DP, DiffSoFar, Buy → Φ

Step 2: DPMemoizationMN(A, DiffSoFar, DP, Buy, m, n, c)

Step 3: return **BACKTRACKERSULT(OPT, BUY, A)**

DPMemoizationMN(A, DiffSoFar, DP, Buy, M, N, c)
Step 1: if N <= 0
DP[M][N] = 0
Return DP[M][N]

Step 2: Current = DP[M][N]

Step 3: PreviousDayProfit = DP[M][N-1] == 0 ? **DPMemoizationMN(A, DiffSoFar, DP, Buy, M, N-1, c) :** DP[M][N-1]

Step 4: previousStockProfit = M <= 0 ? 0 : DP[M-1][N] == 0 ? DPMemoizationMN(A, DiffSoFar, DP, Buy, M-1, N, c) : DP[M-1][N]

Step 5: profitIfWeBuy = DP[m-1][max(0, N-c-2)] == 0 ? DPMemoizationMN(A, DiffSoFar, DP, Buy, m-1, max(0, N-c-2), c) : DP[m-1][max(0, N-c-2)]

Step 6:
currentDiff = profitIfWeBuy - A[M][N-1]
If currentDiff > DiffSoFar[M][0]
DiffSoFar[M][0] = currentDiff
DiffSoFar[M][1] = N-1

Step 7: DP[M][N] = max(PreviousDayProfit, previousStockProfit, A[M][N] + DiffSoFar[M][0])

Step 8: If current != DP[M][N]
buy[M][N] = DiffSoFar[M][1]

Step 9: Return DP[M][N]

MAIN()

Step 1 : Input c, m, n, A[m][n]

Step 2 :

result = Task9A(A[i], m, n, c) // *BOTTOM UP*

Step 3: Return result

Task9B:

Give an iterative bottom-up implementation of Alg9

Pseudo Code:

Task9B(A, m, n, c)
Step 1: DP, DiffSoFar, Buy → Φ
Step 2:

```
For j = 1 to n:
        For i = 1 to m:
                currentDiff = DP[m-1][max(0, j-c-2)] - A[i]j-1]
                If currentDiff > DiffSoFar[i][0]
                        DiffSoFar[i][0] = currentDiff
                        DiffSoFar[i][1] = j-1
                Current = DP[i][j]
                PreviousDayProfit = DP[i][j-1]
                previousStockProfit = i == 0 ? 0 : DP[i-1][j]
                profitIfWeBuy = A[i][j] + DiffSoFar[i][0]

                DP[i][j] = max(PreviousDayProfit, previousStockProfit,profitIfWeBuy
)
                If current != DP[i][j] then
                        Buy[i][j] = DIffSoFar[i][1]
```

Step 3: return BACKTRACKERSULT(OPT, BUY, A)

BACKTRACKERSULT(OPT, BUY, A)
Step 1: i = m-1, j = n-1, result = Φ

Step 2: While(j is not 0)

Current = OPT[i][j

If i is zero and OPT[i][j] == OPT[i][j-1]

j -= 1

else If i > 0 and OPT[i][j] == OPT[i-1][j]

i -= 1

else if OPT[i][j] == OPT[i][j-1]

j -= 1

else

// Transaction occurred

result = result U { { i, A[i][j] - A[i][Buy[i][j]], Buy[i][j], j} }

// Go to profit where we buy from

j = buy[i][j] - c **- 1**

i = m - 1

Step 3: return result

MAIN()

Step 1 : Input c, m, n, A[m][n]

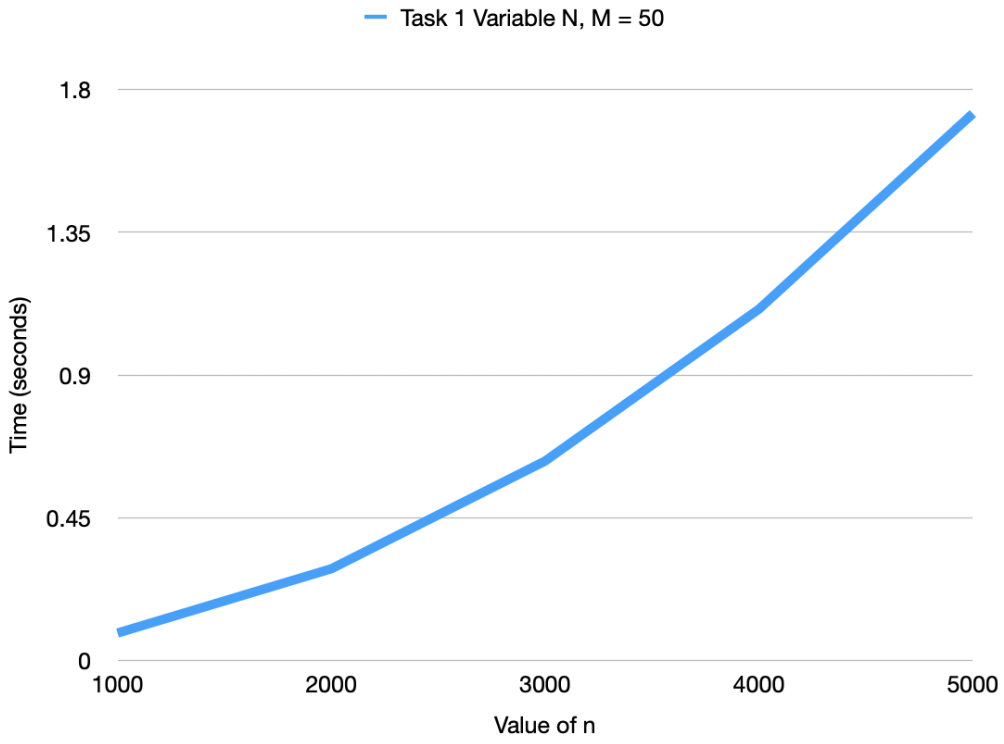
Step 2 :

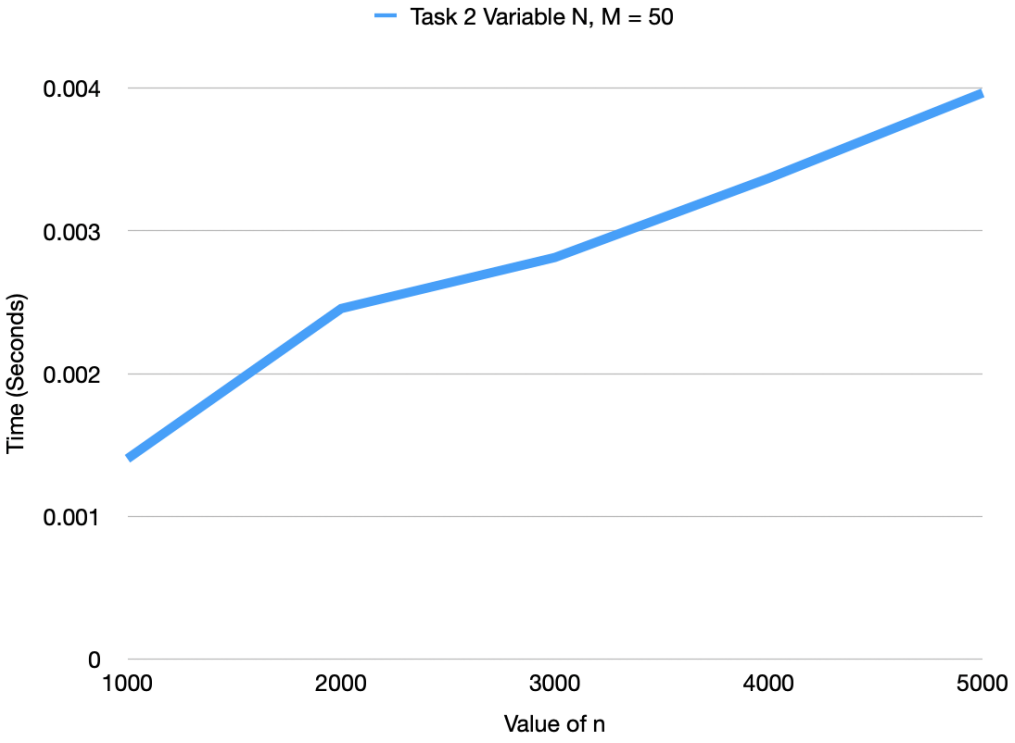
result = Task9B(A[i], m, n, c) // *MEMOIZATION*

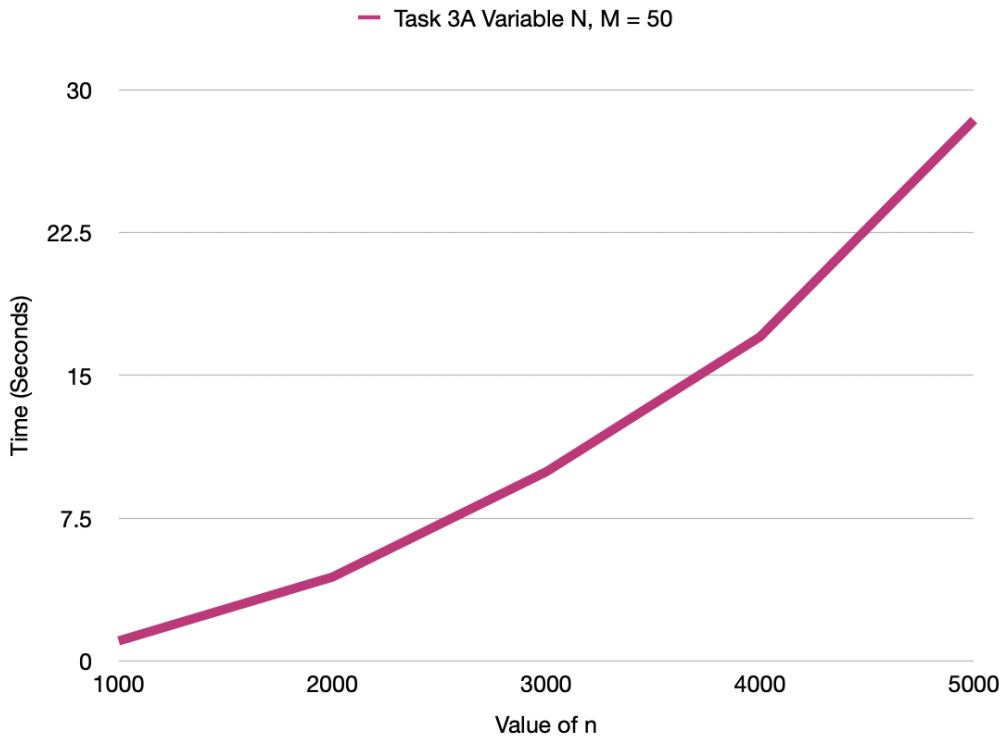
Step 3: Return result

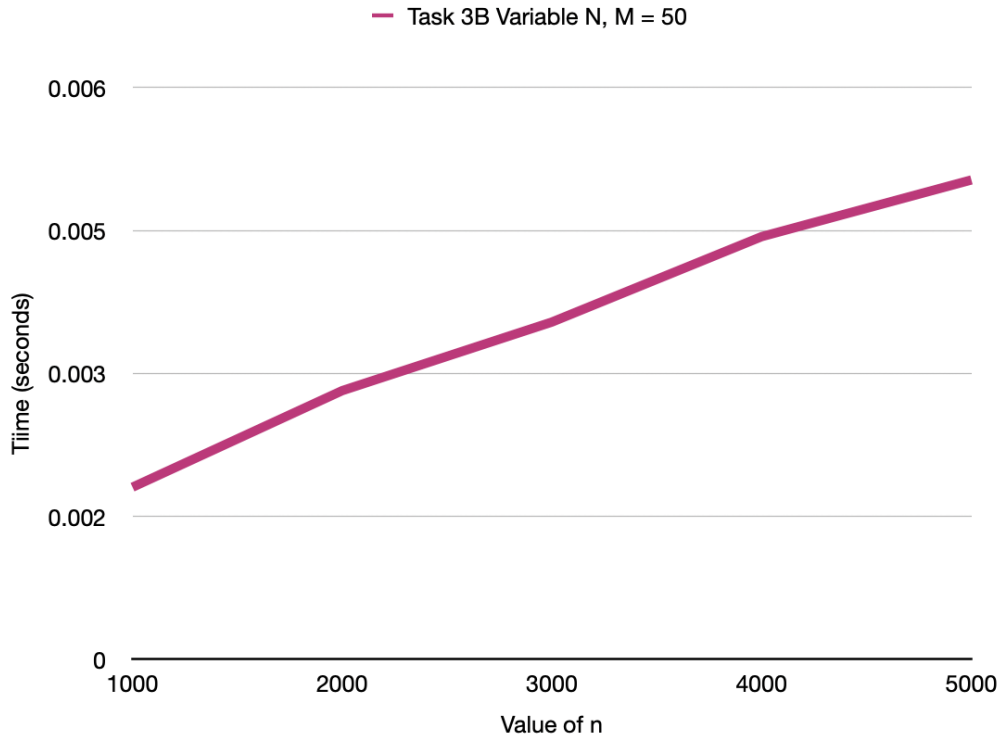
Experimental Comparative Study:

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

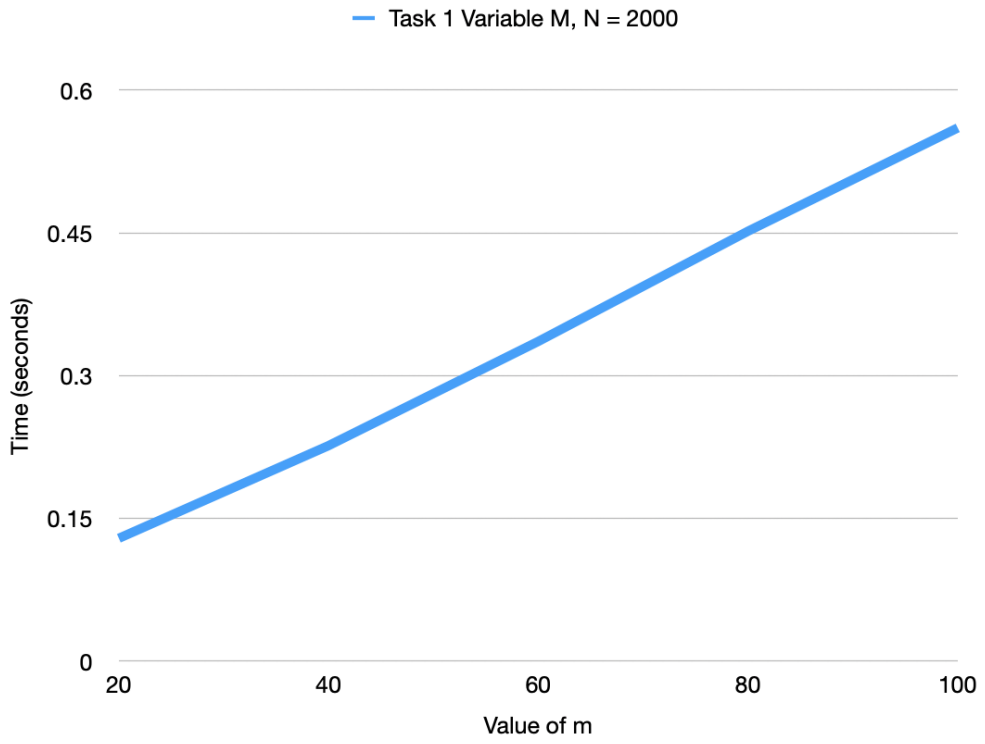


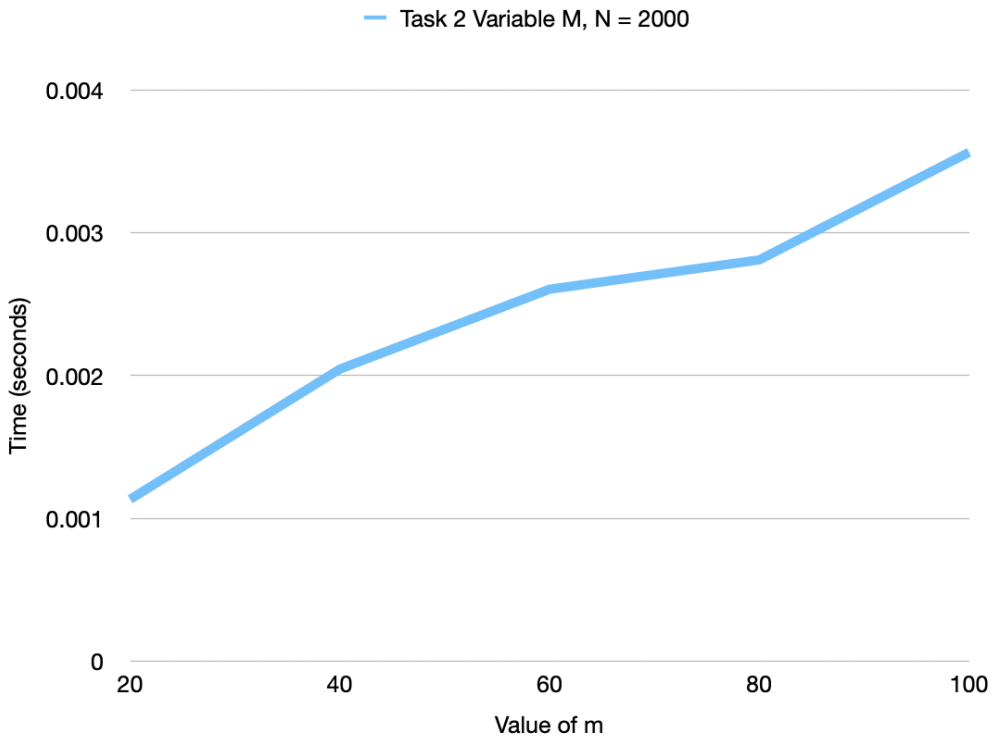


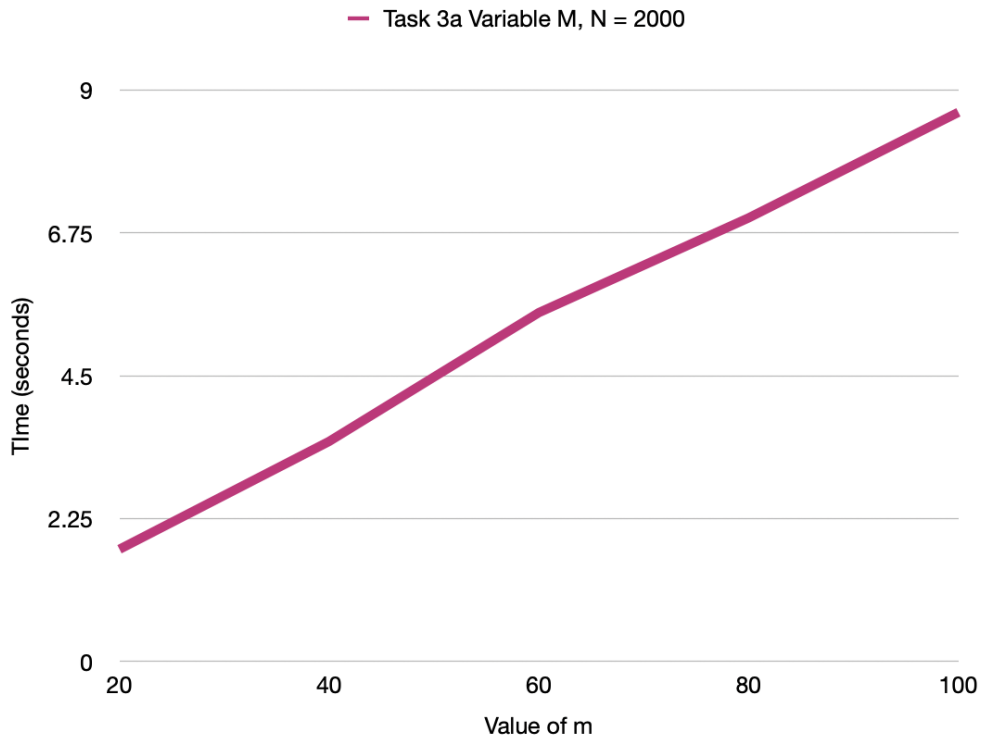

Task 3A Variable N, M = 50
30
22.5
15
7.5
0
Time (Seconds)
1000
2000
3000
4000
5000
Value of n

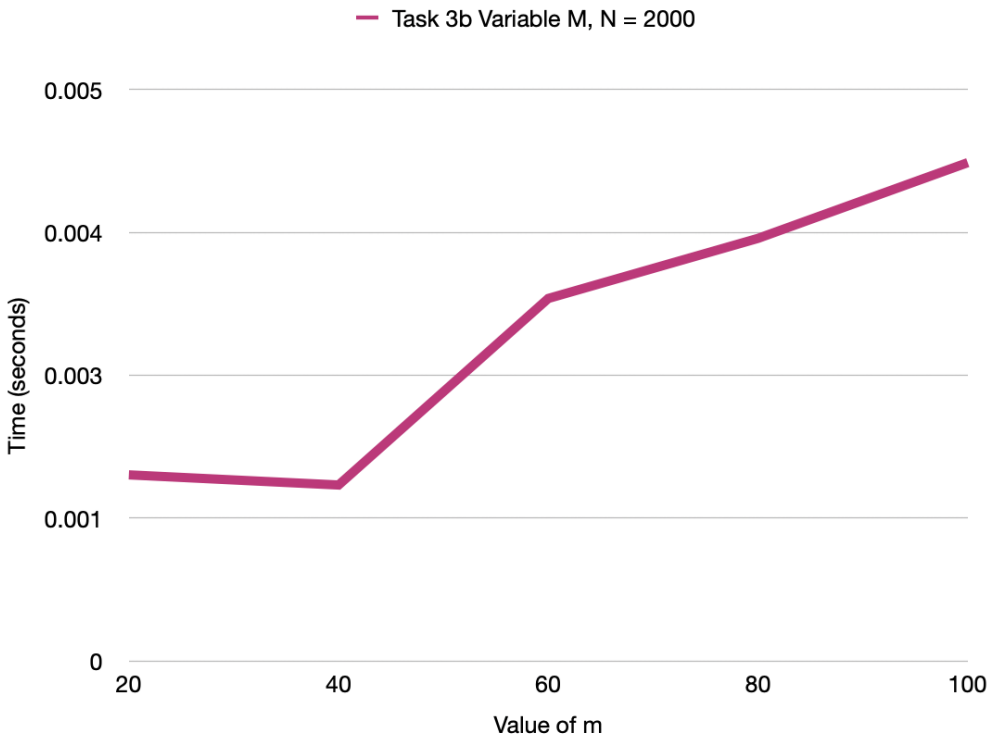

Task 3B Variable N, M = 50
0.006
0.005
0.003
0.002
0
Tiime (seconds)
1000
2000
3000
4000
5000
Value of n

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

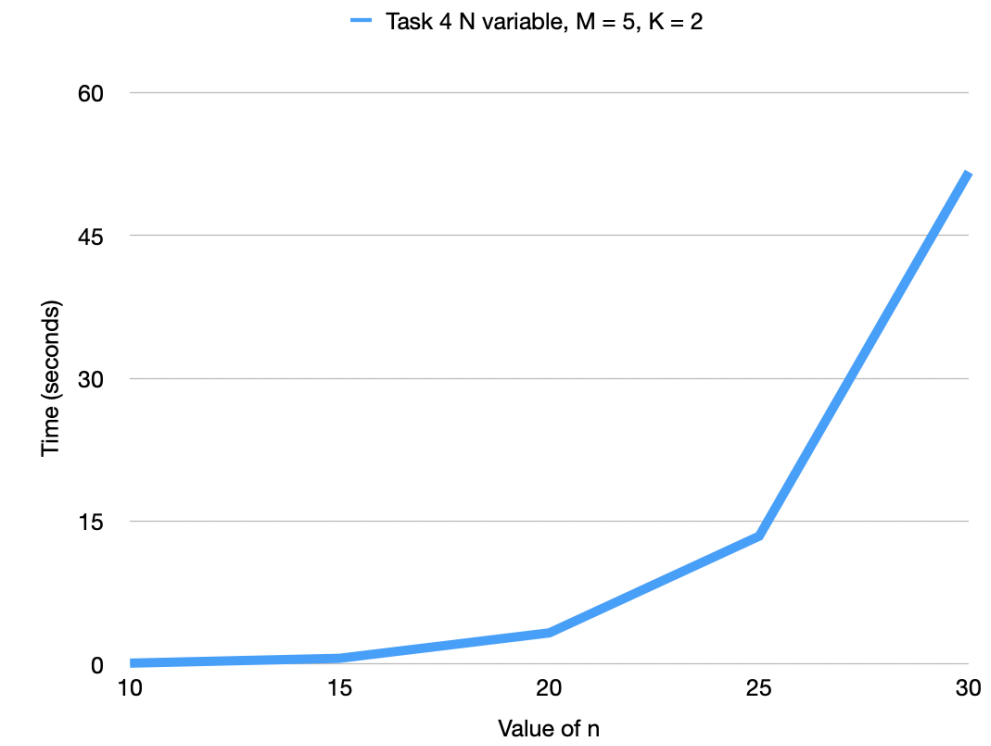


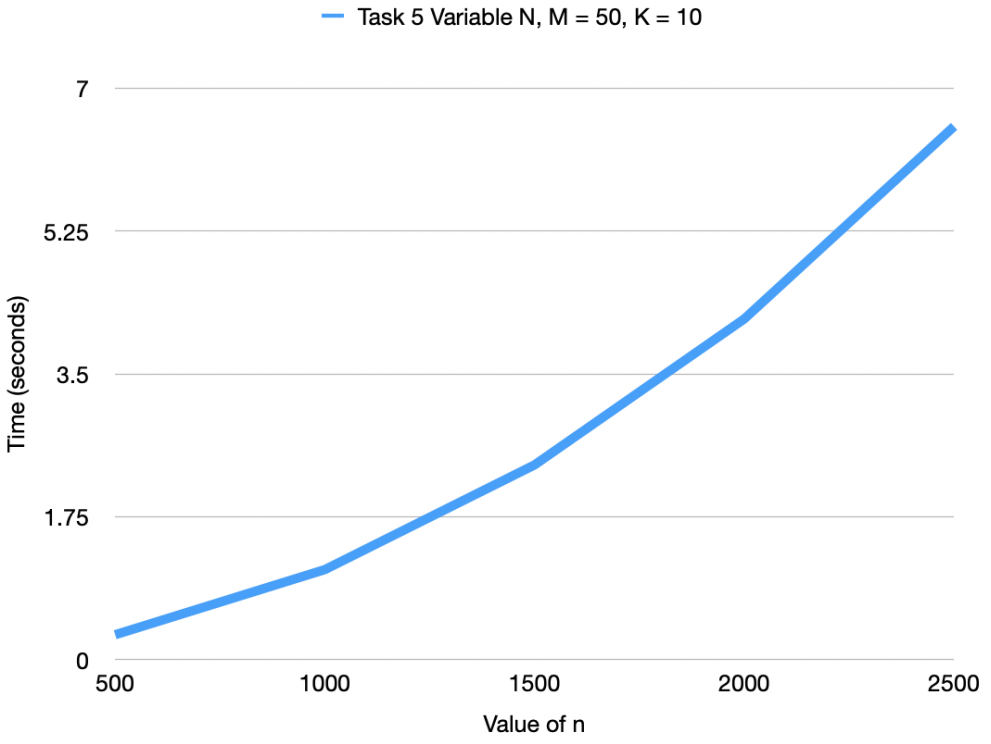


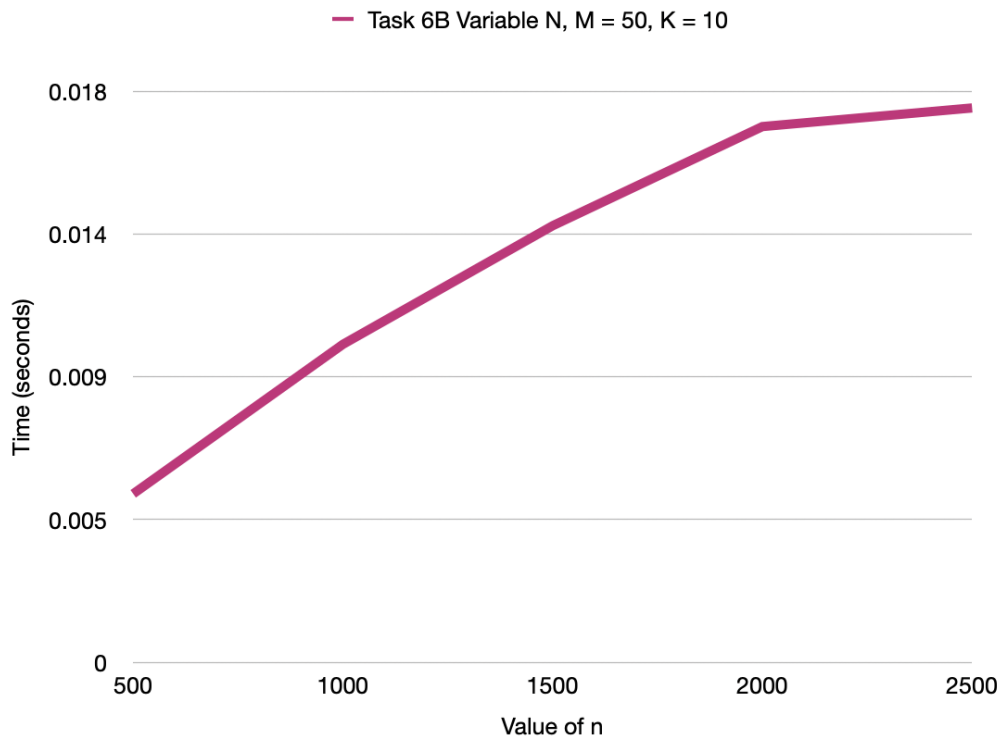

Task 3a Variable M, N = 2000
9
6.75
4.5
2.25
0
TIme (seconds)
20
40
60
80
100
Value of m

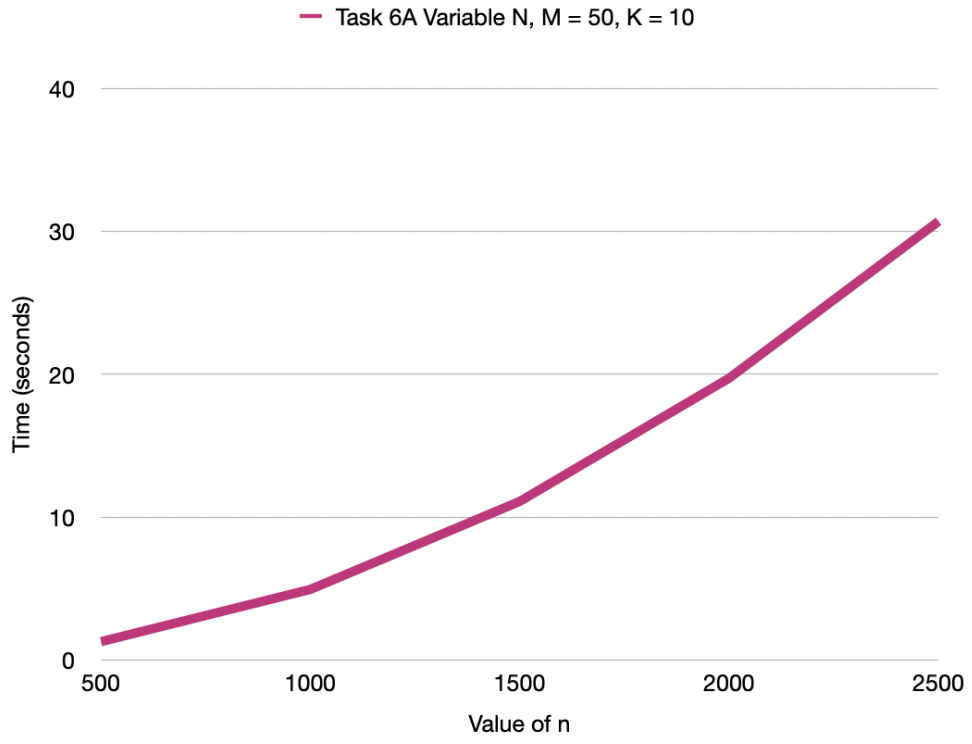

Task 3b Variable M, N = 2000
0.005
0.004
0.003
0.001
0
Time (seconds)
20
40
60
80
100
Value of m

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

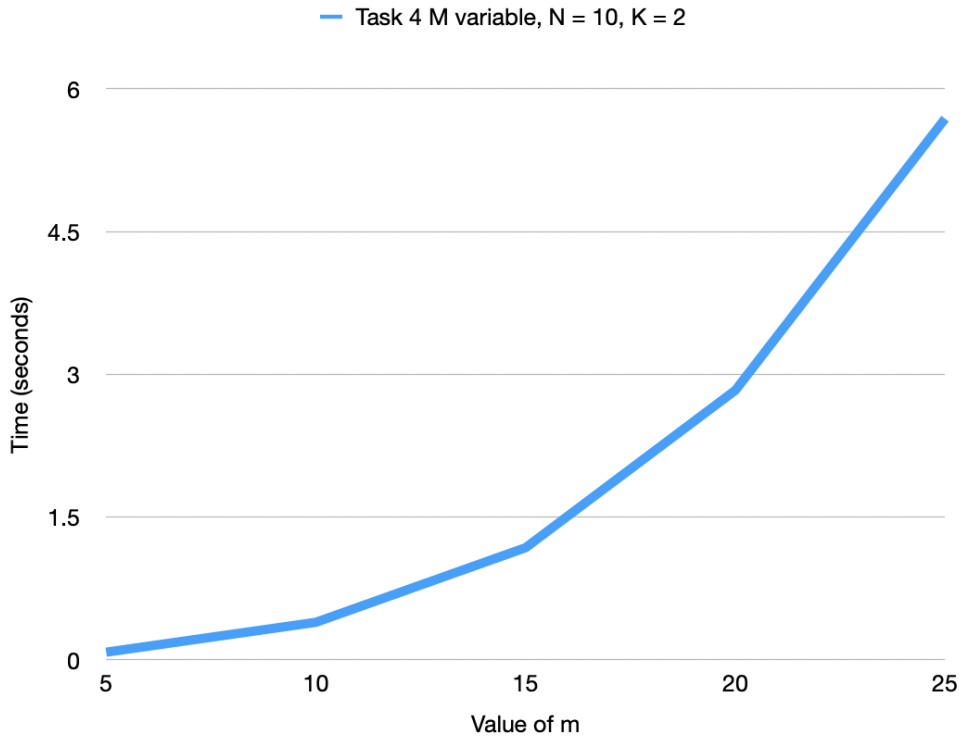


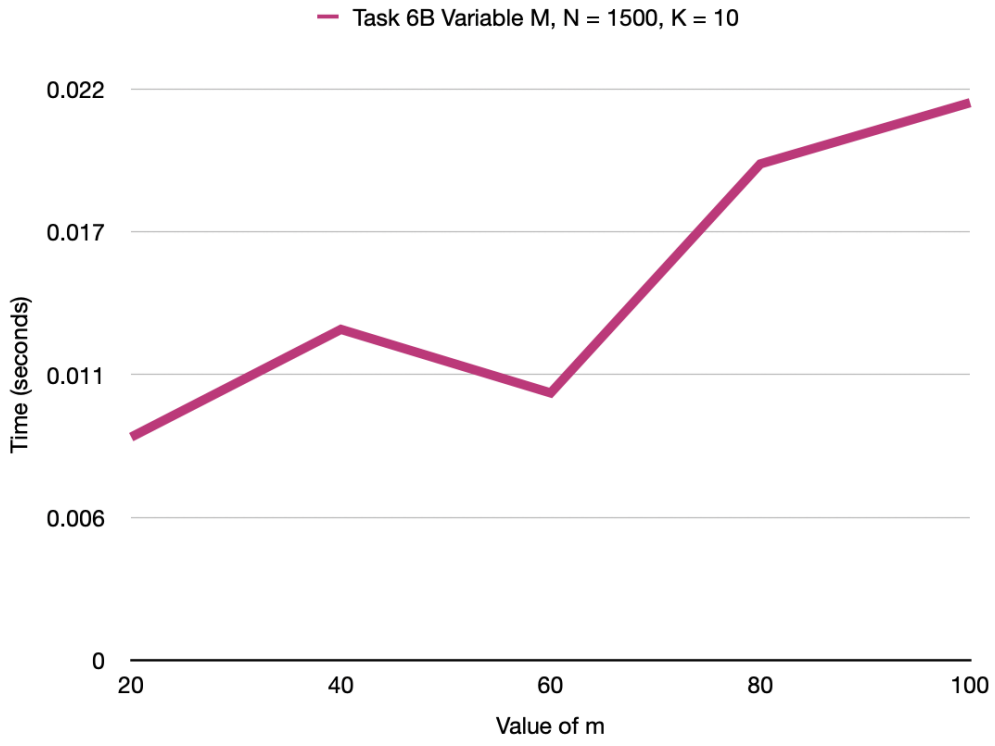


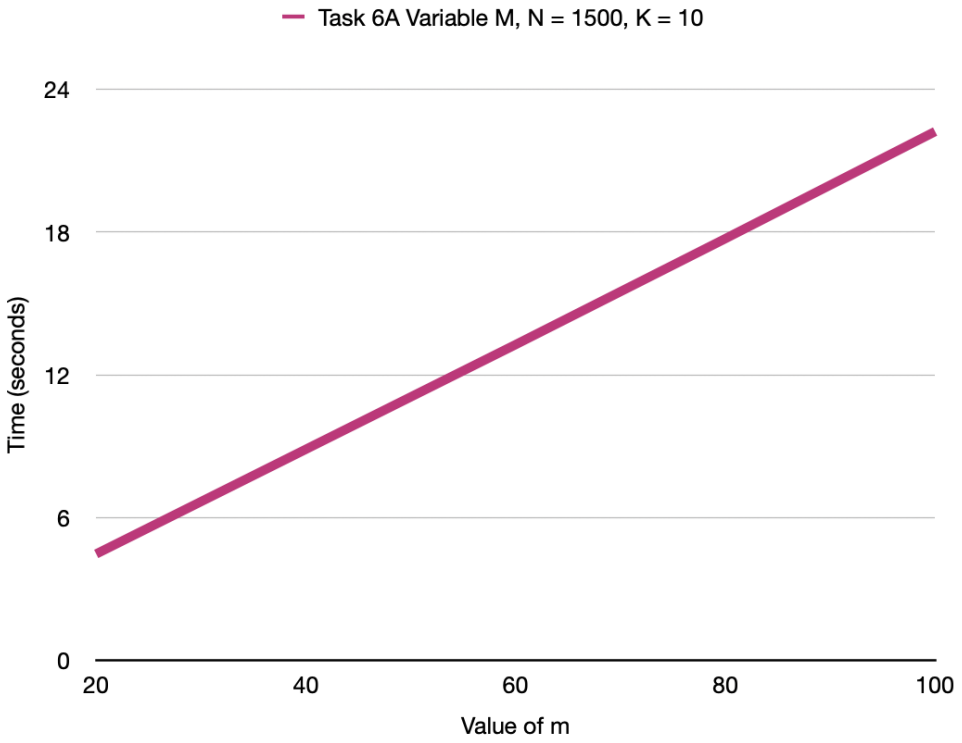

Task 6B Variable N, M = 50, K = 10
0.018
0.014
0.009
0.005
0
Time (seconds)
500
1000
1500
2000
2500
Value of n

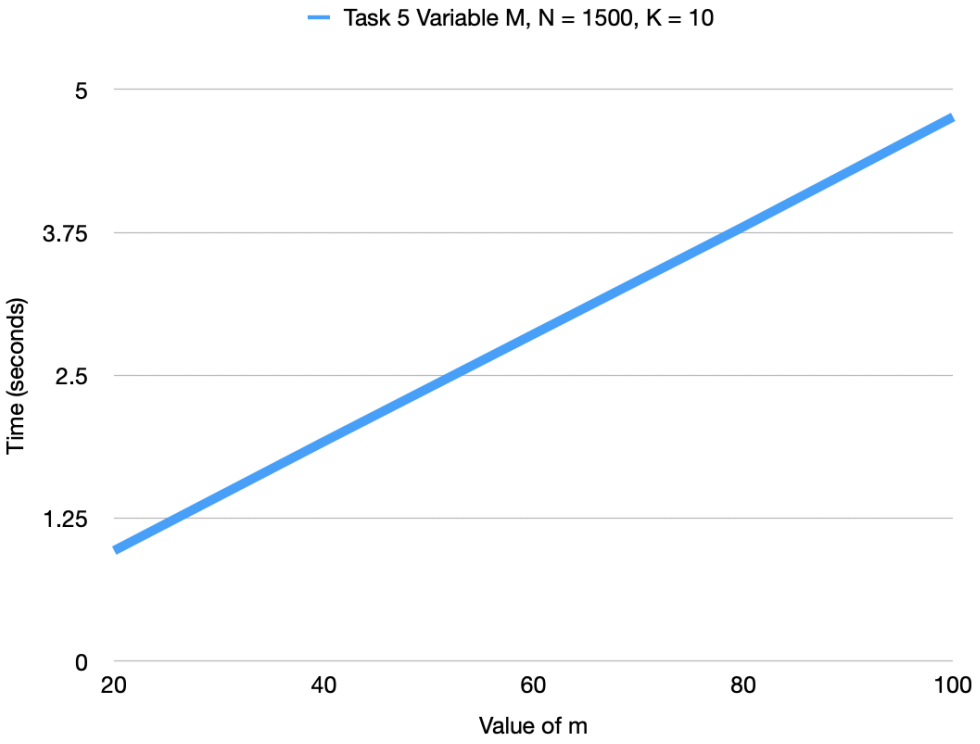

Task 6A Variable N, M = 50, K = 10
40
30
20
10
0
Time (seconds)
500
1000
1500
2000
2500
Value of n

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

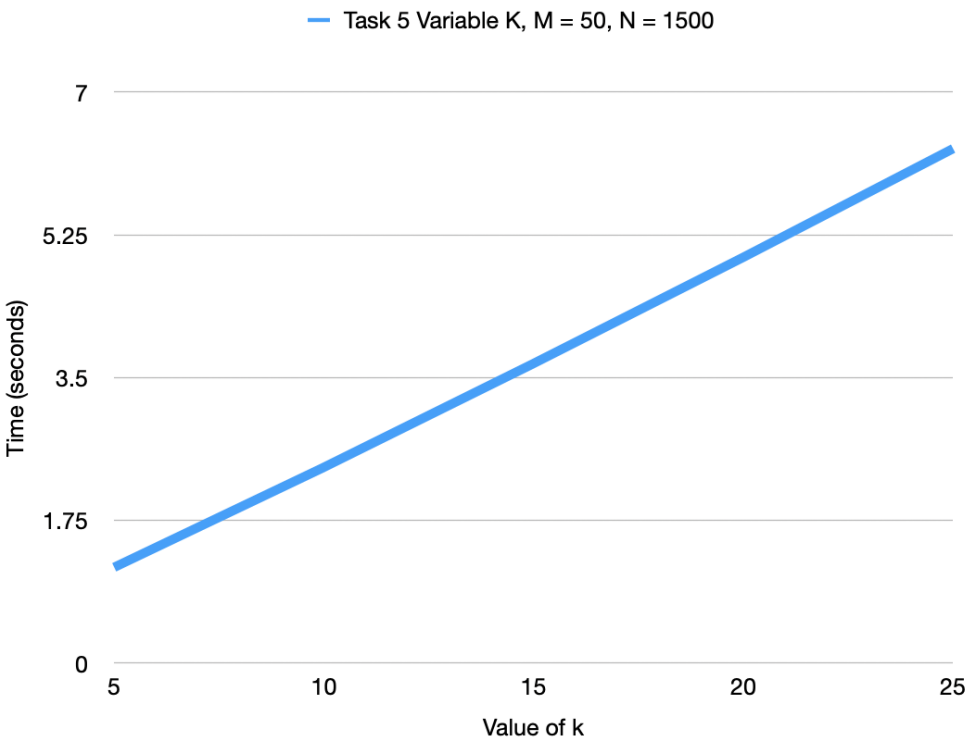


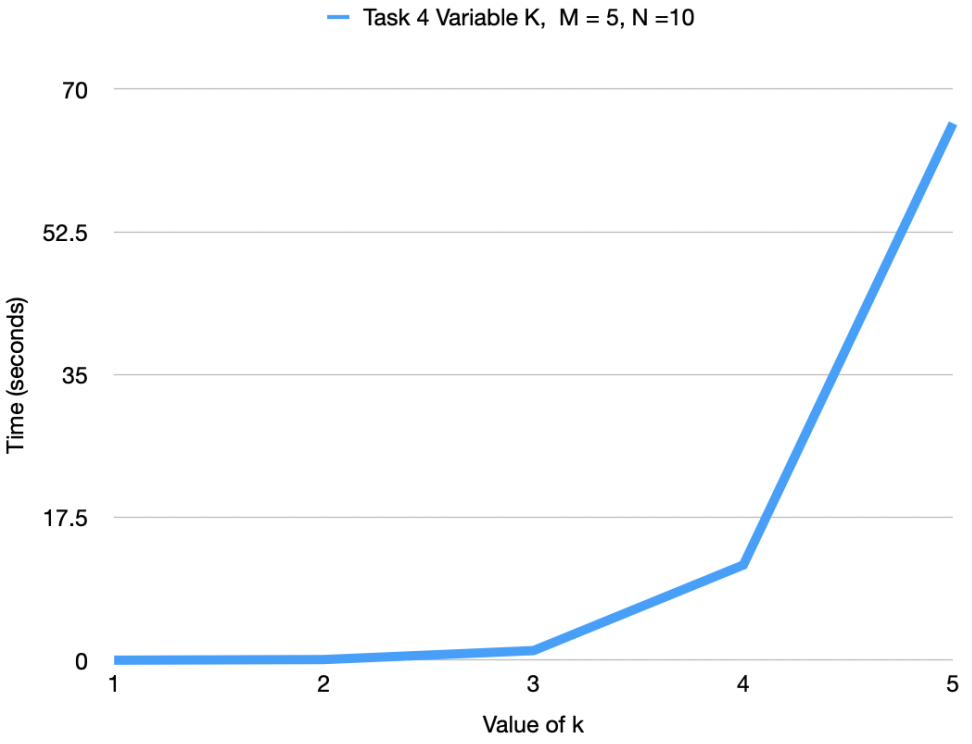


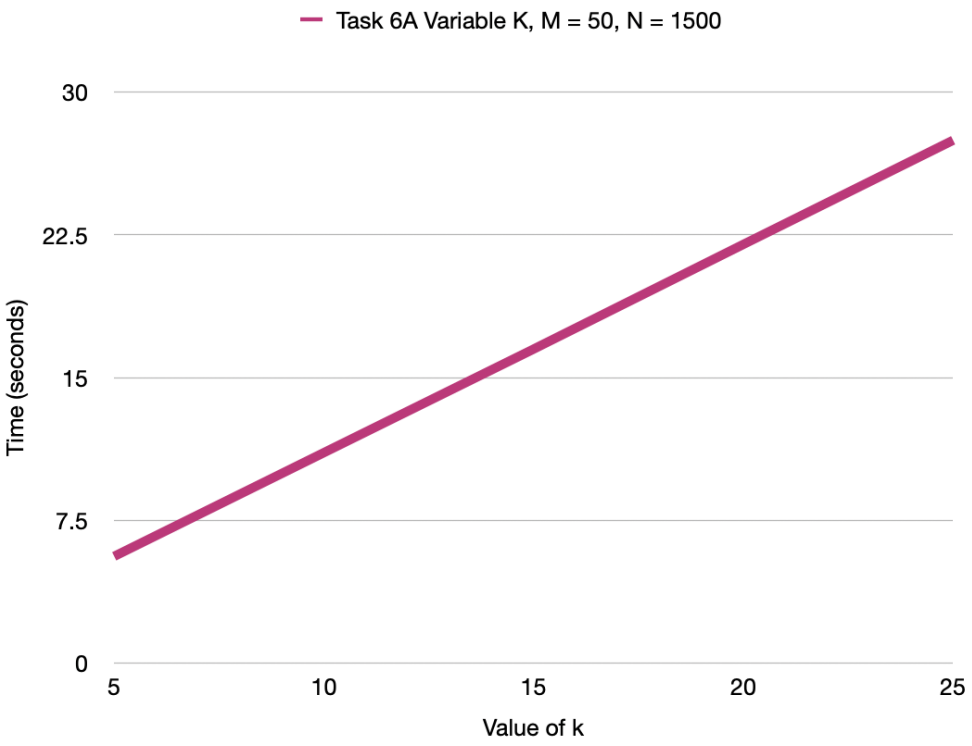

Task 6A Variable M, N = 1500, K = 10
24
18
12
6
0
Time (seconds)
20
40
60
80
100
Value of m

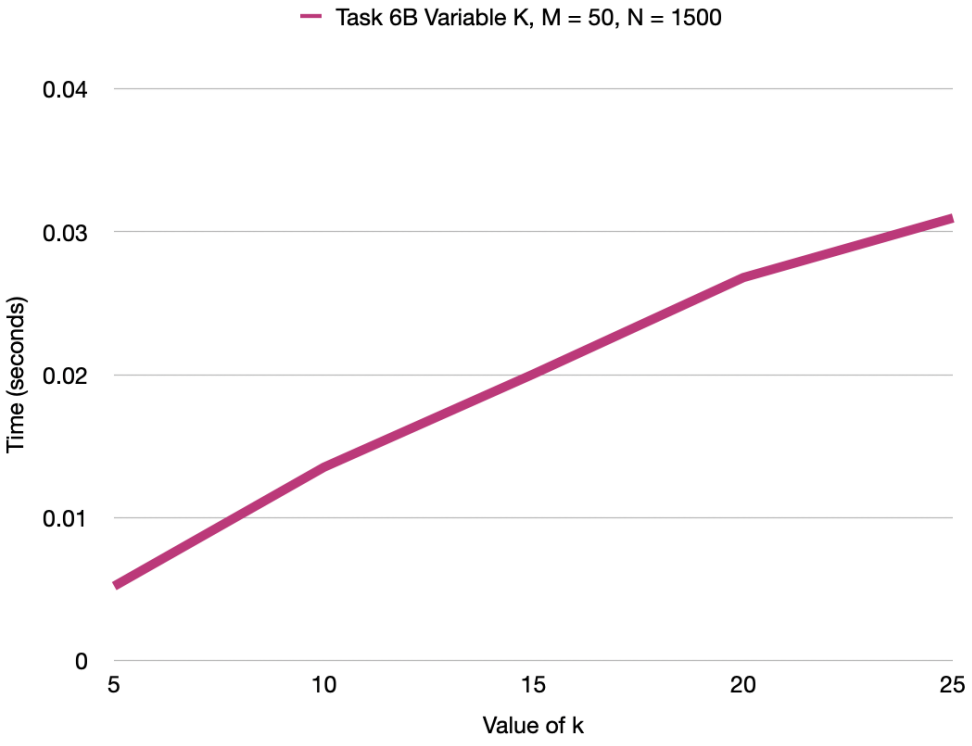

Task 5 Variable M, N = 1500, K = 10
5
3.75
2.5
1.25
0
Time (seconds)
20
40
60
80
100
Value of m

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

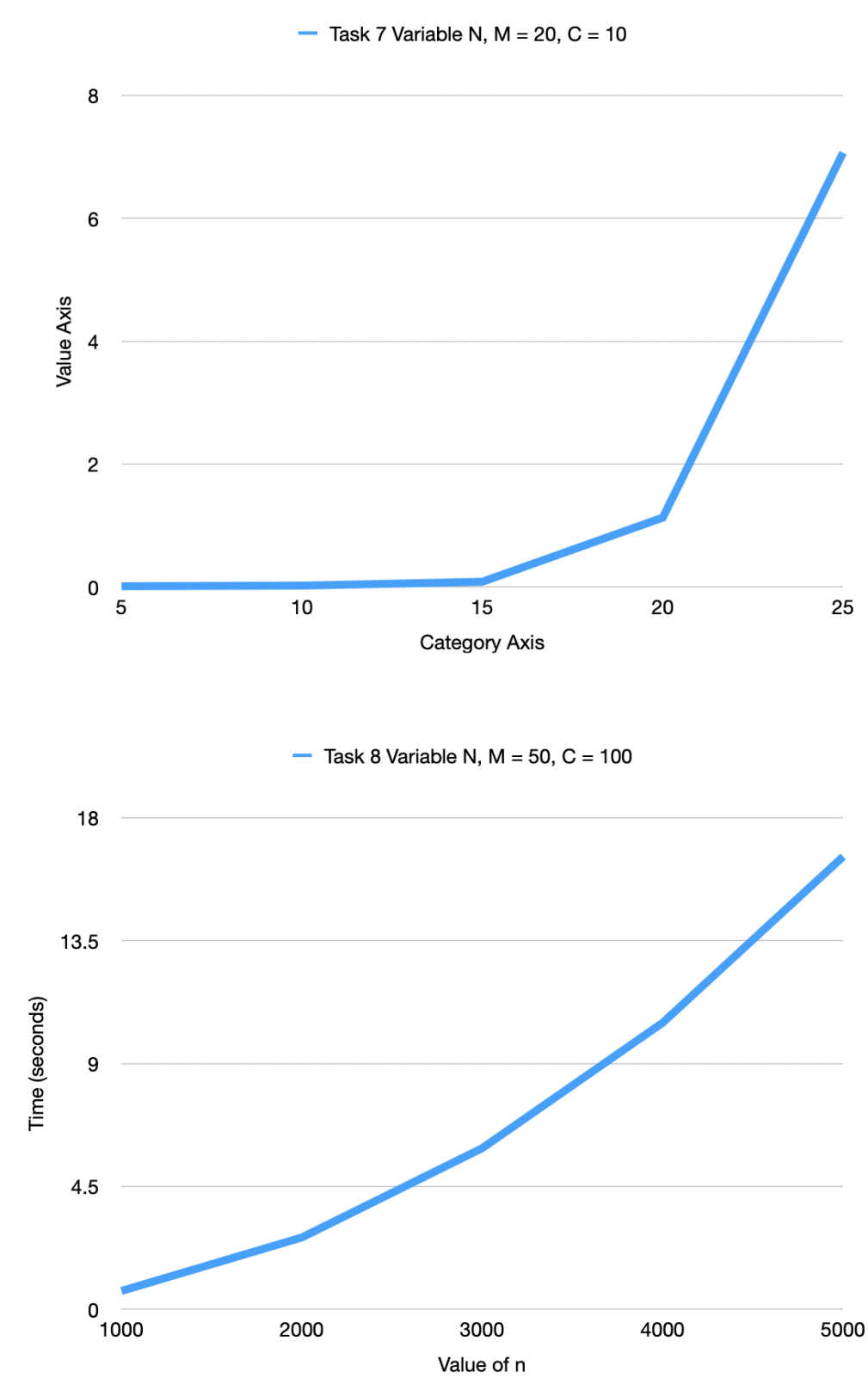

Task 5 Variable K, M = 50, N = 1500
7
5.25
3.5
1.75
0
Time (seconds)
5
10
15
20
25
Value of k

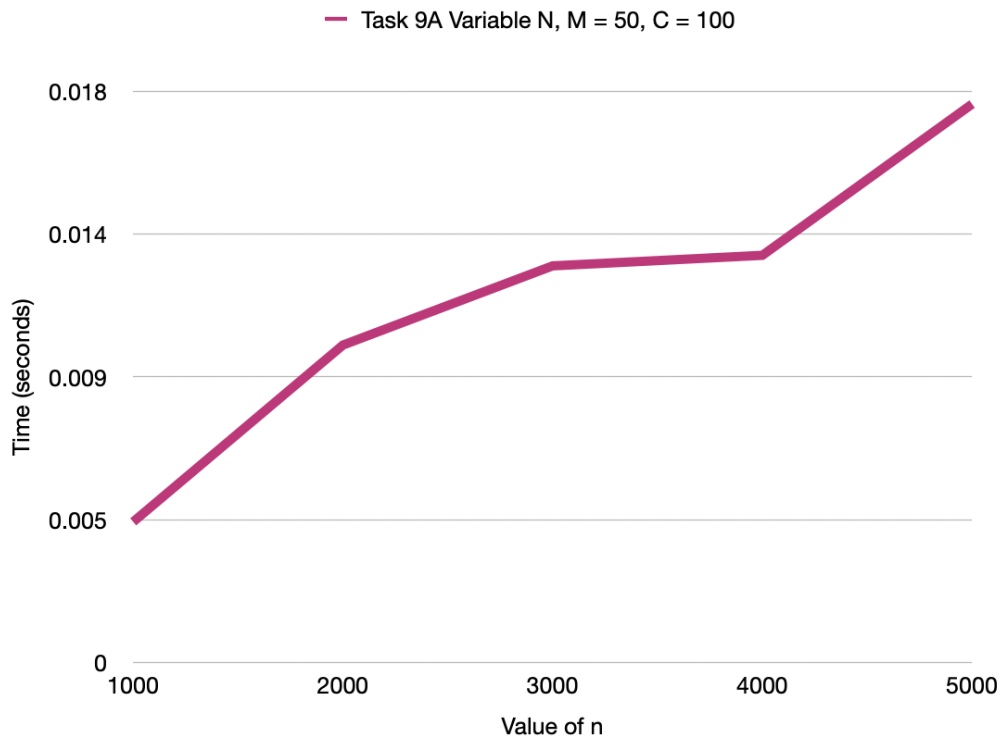

Task 4 Variable K, M = 5, N =10
70
52.5
35
17.5
0
Time (seconds)
1
2
3
4
5
Value of k

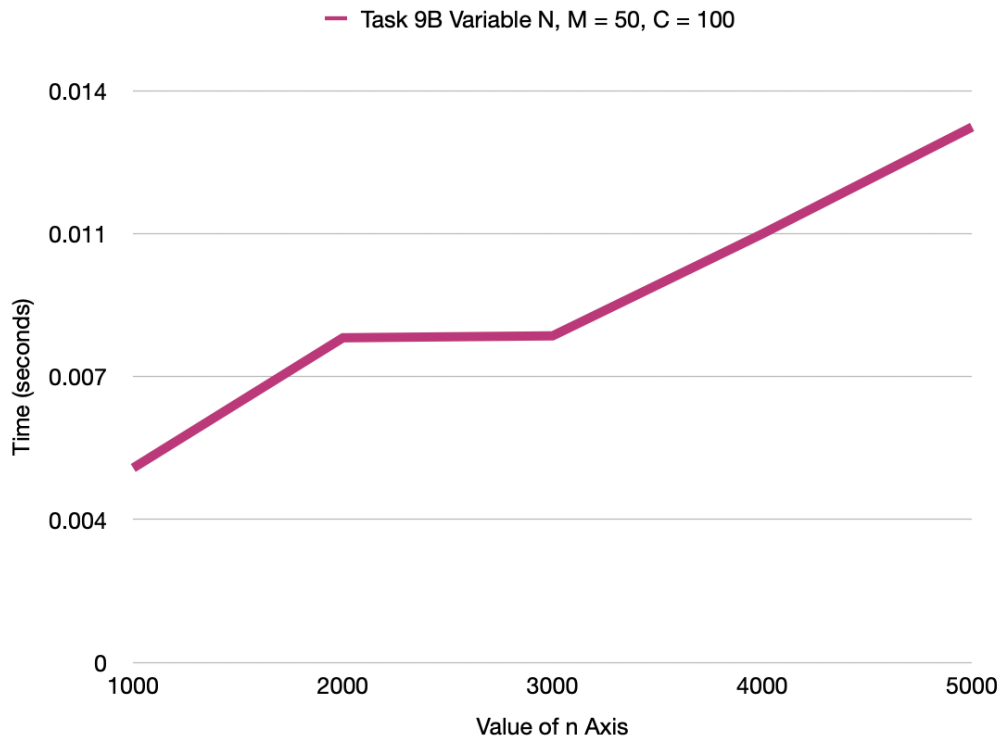

Task 6A Variable K, M = 50, N = 1500
30
22.5
15
7.5
0
Time (seconds)
5
10
15
20
25
Value of k


Task 6B Variable K, M = 50, N = 1500
0.04
0.03
0.02
0.01
0
Time (seconds)
5
10
15
20
25
Value of k

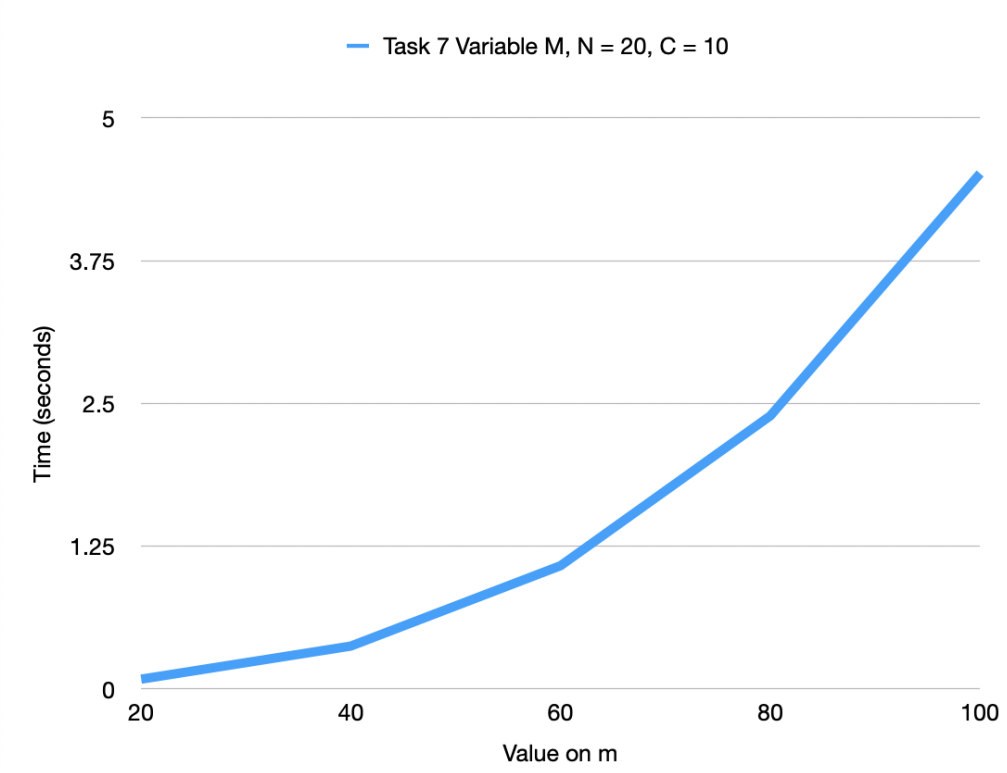
Plot6 Comparison of Task7, Task8, Task9A, Task9B with variable n and fixed m and c.

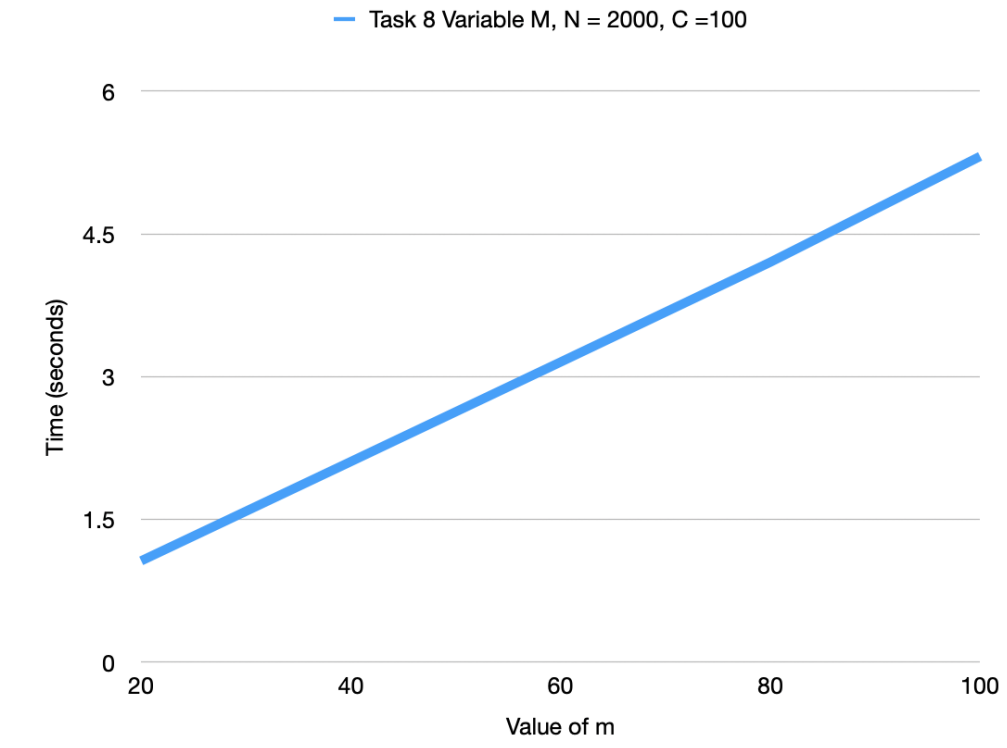


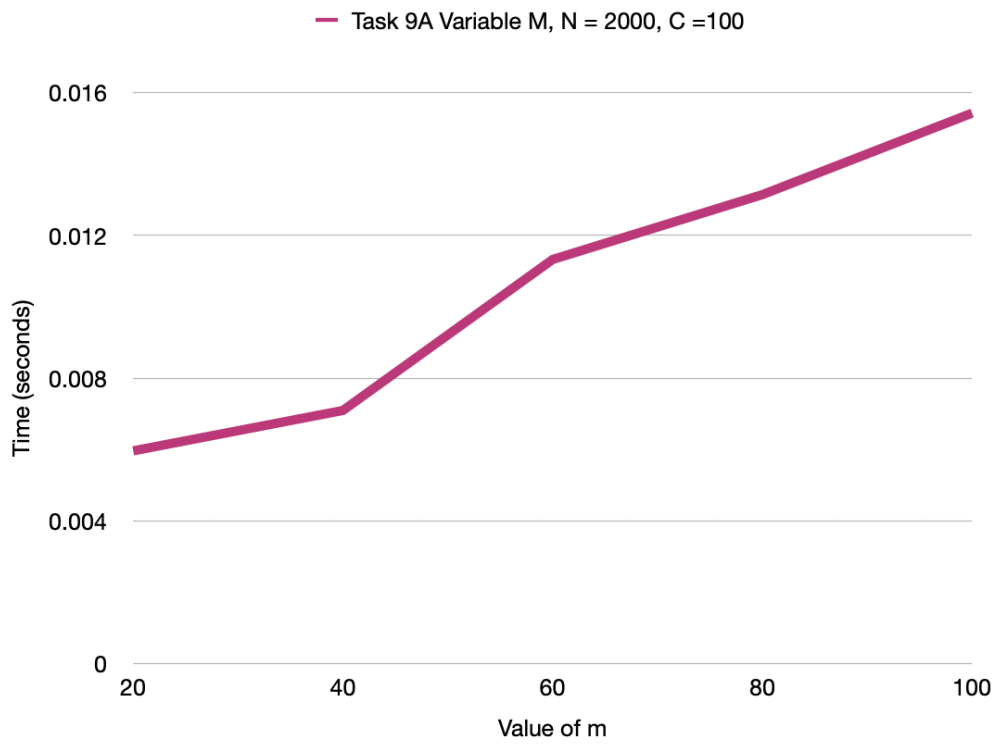

Task 9A Variable N, M = 50, C = 100
0.018
0.014
0.009
0.005
0
Time (seconds)
1000
2000
3000
4000
5000
Value of n

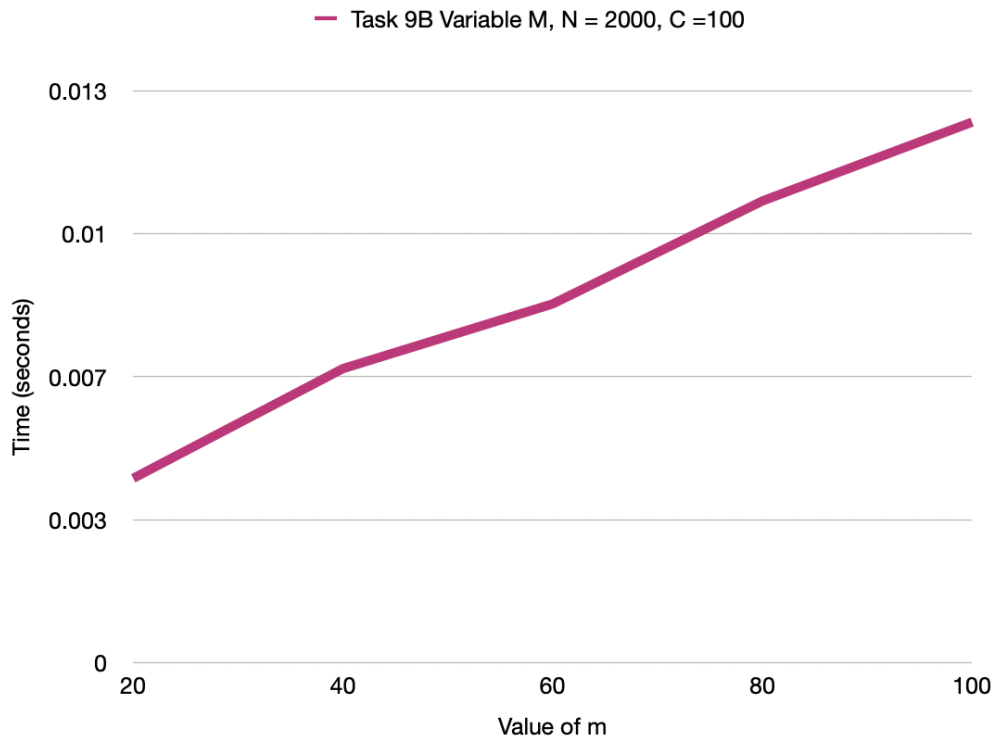

Task 9B Variable N, M = 50, C = 100
0.014
0.011
0.007
0.004
0
Time (seconds)
1000
2000
3000
4000
5000
Value of n Axis

Plot7 Comparison of Task7, Task8, Task9A, Task9B with variable m and fixed n and c






Task 9A Variable M, N = 2000, C =100
0.016
0.012
0.008
0.004
0
Time (seconds)
20
40
60
80
100
Value of m


Task 9B Variable M, N = 2000, C =100
0.013
0.01
0.007
0.003
0
Time (seconds)
20
40
60
80
100
Value of m

TEAM MEMBER CONTRIBUTION:

TEAM MEMBERS:-

1) Dharmam Savani (UFID- 9524 9907)
    - Algorithm 2, 8, 9
2) Dev Patel (UFID- 2328 5413)
    - Algorithm: 1, 5, 6
3) Chintan Acharya (UFID- 6435 4143)
    - Algorithm 3, 4, 7

Here each person was responsible for design, analysis, comparative study and report for respective algorithms. Although distinction about contribution can not be said clearly as we each have brainstormed and came with each solutions together.

CONCLUSION:

- We were tasked to find transactions that yield maximum profit for given constraints for classical stock buy-sell problem but building on that our problem had m stocks.
- We solved the classical buy-sell stock problem with m stocks and a single transaction using brute-force, greedy and dynamic programming.
- Our case study shows that solution obtained with greedy and dynamic programming with a bottom-up approach had far better performance than brute-force. (Given fixed m)
- We see that for this problem, given n is fixed all the solutions with greedy and dynamic programming had almost the same performance.
- Then we were tasked to find k possible transaction where k is arbitrary for the given classical problem but with m stocks.
- We can see the comparison of each algorithms as per the comparative study we did for different variable.
- Then we were tasked for bonus problem where we had infinite amount of transactions with classical buy-sell problem with m stocks and a cool down of c.
- The Algorithms designed as per their complexity is seen to have a same performance as its design.