Stocks Algorithm Project

1. Team Members

- Lauren Newman
  Contributions:
  - Implemented all algorithms in Java
  - Correctness, time, and space analysis for problem 1 and 3
  - Wrote test cases and tested in Thunder
  - All experiments and graph generation
  - Makefile and readme
  - Helped with design for algorithms in problem 1
  - Formatted and contributed to writing of pseudocode
  - Wrote conclusion

- Suhas Katari Chaluva Kumar:
  Contributions:
  - Was responsible for the design of algorithms
  - Developed working prototypes in Python
  - Contributed to the writing of pseudocode of algorithms
  - Was responsible for analysis of algorithms by formulating the correctness, time and space complexities for problems 2 and 3
  - Added comments explaining the non-trivial piece of code across all implementations

2. Design & Analysis of Algorithms

2.1 Algorithm Design

2.1.1 Alg 1 - Problem 1 - Brute Force $\Theta(m * n^2)$

```
P1-BRUTE-FORCE ($A$, $m$, $n$)
----------------------------------------
Init $maxProfit, buyDay, sellDay, stock \leftarrow 0$
FOR $i = 0$ TO $m$
    FOR $j1 = 0$ TO $n$
        FOR $j2 = j1 + 1$ TO $n$
            IF ($A[i][j2]$ - $A[i][j1] > maxProfit$)
                $maxProfit \leftarrow A[i][j2] - A[i][j1]$
                $buyDay \leftarrow j1$
                $sellDay \leftarrow j2$
                $stock \leftarrow i$
RETURN ($stock$, $buyDay$, $sellDay$)
```

2.1.2 Alg 2 -Problem 1 - Greedy Θ(m ∗n)

```
P1-GREEDY ($A$, $m$, $n$)
________________________________________
Init $maxProfit, buyDay, sellDay, stock \leftarrow 0$
FOR $i = 0$ TO $m$
    $min \leftarrow A[i][0]$
    FOR $j = 1$ TO $n$
        IF ($A[i][j] < min$)
            $min \leftarrow A[i][j]$
            $buyDay \leftarrow j$
        ELSE IF ($A[i][j]$ - $min > maxProfit$)
            $maxProfit \leftarrow A[i][j]$ - $min$
            $sellDay \leftarrow j$
            $stock \leftarrow i$
RETURN ($stock$, $buyDay$, $sellDay$)
```

2.1.3 Alg 3 - Problem 1 - Dynamic Θ(m ∗n)

```
P1-DYNAMIC ($A$, $m$, $n$)
________________________________________
$maxProfit \leftarrow 0$
$txn \leftarrow$ empty Object of format <stock, buyDay, sellDay>
FOR $i \leftarrow 0$ TO $m$
    $previousBuyValue \leftarrow -A[i][0]$ // We will assume we buy stock i on day 1
    $previousSellValue \leftarrow 0$
    $currentBuyValue \leftarrow 0$
    $currentSellValue \leftarrow 0$
    $buyDay \leftarrow$ first day
    $SellDay \leftarrow$ first day
    $tempBuyDay \leftarrow$ first day
    FOR $day \leftarrow 1$ TO $n$
        $currentSellValue \leftarrow$ max($previousSellValue$, $previousBuyValue + A[i][day]$)
        IF ($currentSellValue > previousSellValue$)
            $buyDay \leftarrow tempPurchaseDay$
            $saleDay \leftarrow day$
        $currentBuyValue \leftarrow$ max($previousBuyValue$, $-A[i][day]$)
        IF ($currentBuyValue > previousBuyValue$)
            $tempPurchaseDay \leftarrow day$
            $previousBuyValue \leftarrow currentBuyValue$
            $previousSellValue \leftarrow currentSellValue$
    IF ($previousSellValue > maxProfit$)
        $maxProfit \leftarrow previousSellValue$
        $txn \leftarrow$ Transaction(stock: $i$, buyDay: $buyDay$, sellDay: $saleDay$)
```

```
RETURN txn
```

2.1.4 Alg 4 - Problem 2 - Brute Force $\Theta(m * n^{2k})$

P2-BRUTE-FORCE (*A*, *start, k, maxProfit, profitSoFar, allTrans, requiredTrans*)

```
IF ($k = 0$)
    IF ($profitSoFar > maxProfit$)
        $requiredTrans \leftarrow allTrans$
        $maxProfit \leftarrow profitSoFar$
    RETURN $profit$, $requiredTrans$
$m \leftarrow$ height of $A$, $n \leftarrow$ length of $A$
FOR $j1 = start$ TO $n$
    FOR $j2 = j1 + 1$ TO $n$
        $maxSoFar$, $buyDay$, $sellDay$, $stock \leftarrow 0$
        FOR $i = 0$ TO $m$
            IF ($A[i][j2]$ - $A[i][j1] > maxSoFar$)
                $maxSoFar \leftarrow A[i][j2]$ - $A[i][j1]$
                $buyDay \leftarrow j$
                $sellDay \leftarrow j2$
                $stock \leftarrow i$
            $allTrans[k] \leftarrow [stock, buyDay, sellDay]$
            $maxProfit$, $requiredTrans \leftarrow$ P2-BRUTE-FORCE($A$, $j2$, $k$ - 1, $maxProfit$, $profitSoFar +$
                                                    $maxSoFar$, $allTrans$, $requiredTrans$)
RETURN ($stock$, $buyDay$, $sellDay$) for each transaction in $requiredTrans$
```

2.1.5 Alg 5 - Problem 2 - Dynamic 1 $\Theta(m * n^2 * k)$

P2-DYNAMIC1 (*A*, *k*)

```
$m \leftarrow$ height of $A$,   $n \leftarrow$ length of $A$
$dpProfit \leftarrow k$+1 by n integer matrix
$dpTxns \leftarrow k$+1 by n Transaction matrix // format: [stockIdx, buyIdx, sellIdx, dpProfit<i, j>]
FOR t $\leftarrow$ 0 TO $k + 1$
    FOR $j2 \leftarrow 1$ TO $n$
        $totProfit \leftarrow 0$
        $txn \leftarrow$ null Transaction
        FOR $j1 \leftarrow 0$ TO $j2$
            FOR $i \leftarrow 0$ TO $m$
                $currProfit \leftarrow A[i][j2]$ - $A[i][j1] + dpProfit[t$ - 1$][j1]$
                IF ($currProfit > totProfit$)
                    $totProfit \leftarrow currProfit$
                    $txn \leftarrow$ Transaction(stock: $i$, buyDay: $j1$, sellDay: $j2$, dpProfit: ($t$ - 1, $j1$))
        IF (totProfit > dpProfit[t][j2 - 1])
            dpProfit[t][j2] ← totProfit
            dpTxns[t][j2] ← txn
        ELSE
            dpProfit[t][j2] ← dpProfit[t][j2 - 1]
            dpTxns[t][j2] ← dpTxns[t][j2 - 1]
optimal ← empty list of transactions
last ← last item in dpTxns
WHILE last
    Add last to optimal
    last ← dpTxn[last.dpProfit[0]][last.dpProfit[1]];
RETURN (stock, buyDay, sellDay) for each transaction in optimal
```

2.1.6 Alg 6 - Problem 2 - Dynamic 2 Θ(m ∗ n * k)

```
P2-DYNAMIC2 (A, k)
______________________________________
m ← height of A,   n ← length of A
dpProfit ← k+1 by n integer matrix
dpTxns ← k+1 by n Transaction matrix // format: [stockIdx, buyIdx, sellIdx, dpProfit<i, j>]
FOR t ← 0 TO k + 1
    maxDiff ← first column of A * -1
    maxDiffDay ← empty array of size m
    FOR j ← 1 TO n
        totProfit ← 0
        txn ← null Transaction
        FOR i ← 0 TO m
            currProfit ← A[i][j] + maxDiff[i]
            IF (currProfit > totProfit)
                totProfit ← currProfit
                txn ← Transaction(stock: i, buyDay: j1, sellDay: j2, dpProfit: (t - 1, j1))
            currDiff ← dpProfit[t - 1][j] - A[i][j]
            IF (currDiff > maxDiff[i])
                maxDiff[i] ← currDiff
                maxDiffDay[i] ← j

        IF (totProfit > dpProfit[t][j - 1])
            dpProfit[t][j] ← totProfit
            dpTxns[t][j] ← txn
        ELSE
            dpProfit[t][j] ← dpProfit[t][j - 1]
            dpTxns[t][j] ← dpTxns[t][j - 1]
optimal ← empty list of transactions
last ← last item in dpTxns
```

```
WHILE last
    Add last to optimal
    last ← dpTxn[last.dpProfit[0]][last.dpProfit[1]];
RETURN (stock, buyDay, sellDay) for each transaction in optimal
```

2.1.7 Alg 7 - Problem 3 - Brute Force $\Theta(m * 2^n)$

```
P3-BRUTE-FORCE (A, c, i, j, buyFlag, txns)
________________________________________

m ← height of A,   n ← length of A
IF (i >= m OR j >= n)
  RETURN txns, 0
profit ← 0
mTxns ← txns
IF (buyFlag)
  FOR k ← i TO m
    notBuyTxns, notBuyProfit ← P3-BRUTE-FORCE(A, c, k, j + 1, true, txns)
    buyTxns, buyProfit ←P3-BRUTE-FORCE(A, c, k, j + 1, false, buyTxns + Transaction(stock: k, buyDay:
                                                                                  j, sellDay: 0)
    buyProfit ← buyProfit - A[k][j]
    IF (buyProfit > profit)
      mTxns ← buyTxns     profit ← buyProfit
    IF (notBuyProfit > profit)
      mTxns ← notBuyTxns     profit ← notBuyProfit
ELSE
   notSellTxns, notSellProfit ← P3-BRUTE-FORCE(A, c, i, j + 1, false, txns)
  last ← last transaction in txns
  sellTxns, sellProfit ← P3-BRUTE-FORCE(A, c, 0, j + c, true, txns - last + Transaction(stock: i, buyDay:
                                                                          last.buyDay, sellDay: j);
  sellProfit ← sellProfit + A[i][j]
  IF (sellProfit > profit)
    mTxns ← sellTxns     profit ← sellProfit
  IF (notSellProfit > profit)
    mTxns ← notSellTxns     profit ← notSellProfit
RETURN mTxns, profit
```

2.1.8 Alg 8 - Problem 3 - Dynamic 1 $\Theta(m * n^2)$

```
P3-DYNAMIC1 (A, c)
________________________________________

m ← height of A,   n ← length of A
dp ← integer array of length n
dpTxns ← Transaction array of length n
FOR k ← 0 TO m
```

```
    $profit \leftarrow A[k][1] - A[k][0]$
    IF ($profit > dp[1]$)
        $dp[1] \leftarrow profit$
        $dpTxns[1] \leftarrow$ Transaction(stock: $k$, buyDay: 0, sellDay: 1, prevIdx: 0)
FOR $i = 2$ TO $n$
    $dp[i] \leftarrow dp[i - 1]$
    $dpTxns[i] \leftarrow dpTxns[i - 1]$
    FOR $j \leftarrow 0$ TO $i$
        $prev \leftarrow 0$
        $prevTxn \leftarrow 0$
        IF ($j >= c$)
            $prev \leftarrow dp[j - c]$;
            $prevTxn \leftarrow j - c$;
        FOR $k \leftarrow 0$ TO $m$
            $profit \leftarrow prev + A[k][i] - A[k][j]$;
            IF ($profit > dp[i]$)
                $dp[i] \leftarrow profit$
                $dpTxns[i] \leftarrow$ Transaction(stock: $k$, buyDate: $j$, sellDay: $i$, prevIdx: $prevTxn$)
$optimal \leftarrow$ empty list of transactions
$last \leftarrow$ last item in $dpTxns$
WHILE $last$
    Add $last$ to $optimal$
    $last \leftarrow dpTxn[last.$prevIdx$]$;
RETURN ($stock$, $buyDay$, $sellDay$) for each transaction in $optimal$
```

2.1.9 Alg 9 - Problem 3 - Dynamic 2 Θ(m ∗ n)

```
P3-DYNAMIC2 ($A$, $c$)
--------------------------------------------
$m \leftarrow$ height of $A$,   $n \leftarrow$ length of $A$
$dp \leftarrow$ integer array of length $n$
$dpTxns \leftarrow$ Transaction array of length $n$
$dpK \leftarrow$ integer matrix of $m$ by $n$
$dpKTxns \leftarrow$ Transaction matrix of $m$ by $n$
$maxDiff \leftarrow$ array of -infinity of length $m$
$maxDiffDay \leftarrow$ integer matrix of $m$ by 2
FOR $i \leftarrow 0$ TO n
    IF ($i < c$)
        FOR $k \leftarrow 0$ TO $m$
            IF ($-A[k][i] > maxDiff[k]$)
                $maxDiff[k] \leftarrow -stocks[k][i]$
                $maxDiffDay[k] \leftarrow [i, 0]$
    IF ($i = 1$)
        FOR $k \leftarrow 0$ TO $m$
```

```
            profit ← A[k][1] - A[k][0]
            IF (profit > dp[1])
                dpTxns[1] ← Transaction(stock: k, buyDay: 0, sellDay: 1, prevIdx: 0)
                dp[1] ← profit
            IF (profit > dpK[k][1])
                dpKTxns[k][1] ← Transaction(stock: k, buyDay: 0, sellDay: 1, prevIdx: 0)
                dpK[k][1] ← profit
    ELSE IF (i > 1)
        FOR k ← 0 TO m
            IF (maxDiff[k] + A[k][i] > dpK[k][i - 1])
                dpK[k][i] ← maxDiff[k] + A[k][i]
                dpKTxns[k][i] ← Transaction(stock: k, buyDay: maxDiffDay[k][0], sellDay: i, prevIdx:
                                                                                maxDiffDay[k][1])
            ELSE
                dpK[k][i] ← dpK[k][i - 1]
                dpKTxns[k][i] ← dpKTxns[k][i - 1]
            IF (dpK[k][i] > dp[i])
                dp[i] ← dpK[k][i]
                dpTxns[i] ← dpKTxns[k][i]
            IF (dp[i - c] - A[k][i] > maxDiff[k])
                maxDiff[k] ← dp[i - c] - A[k][i]
                maxDiffDay[k] ← [i, i - c]
optimal ← empty list of transactions
last ← last item in dpTxns
WHILE last
    Add last to optimal
    last ← dpTxn[last.prevIdx];
RETURN (stock, buyDay, sellDay) for each transaction in optimal
```

2.2 Analysis

2.2.1 Alg 1 - Problem 1 - Brute Force Θ(m $*$n^2)

This algorithm compares every pair of days for each stock where j1 < j2. When it finds a value that is greater than the current max profit, it replaces the information stored for the max profit. It returns the stock, buy day, and sell day of the transaction that generates the most profit.

Correctness

- We use **Proof by Contradiction**, to prove the correctness of this algorithm.
- Let's consider that our algorithm output's **<x, p, i, j>** where **x** is the name of the stock that generates the maximum profit **p** when bought on day **i** and sold on day **j**.
- Let's assume that there is an output **<y, p',i',j'>** where **y** is the name of the stock that generates the maximum profit **p' > p**, when bought on day **i'** and sold on day **j'.**

- Our algorithm considers every stock available, and profit produced for combinations of day from day 1 to day n and updates only when the **newProfit > oldProfit**.
- This means that our algorithm encountered stock **x** with profit **p** and compared it with stock **y** of profit **p'** and determined that **p > p'** since the final output is **x** with profit **p**.
- But **y** with profit **p'** is assumed to be the transaction with maximum profit, **a contradiction**.

Time Complexity

- We need to run a for loop from 0 to m to check the profits that can be generated by all stocks available. The would result in O(m)
- We then run two more for loops, one from 0 to n, and other from i+1 to n (where i tracks the outer for loop) to find every possible buy and sell day combination and this results in O(n*n)
- Now, since these loops are nested, we can write O(m) * O(n*n).
- Therefore, the time complexity is ***$O(m * n^2)$***

Space Complexity

- The only thing being stored is the final solution consisting of buyDay, sellDay, stockName and maxProfit
- The space is not dependent on *m* or *n*
- Therefore, the space complexity is constant, i.e. ***O(1)***

2.2.2 Alg 2 - Problem 1 - Greedy $\Theta(m * n)$

The greedy algorithm maintains a minimum cost for each stock. It will greedily purchase on the minimum cost day and trade on the day that has the biggest difference between its value and the minimum. It returns the stock, buy day, and sell day of the transaction that generates the most profit.

Correctness

- We can **prove it intuitively and use method of contradiction**.
- Let's say that **p** is the maximum profit given by our algorithm as a result of transacting stock **x** by buying on day **i** and selling on day **j**.
- Let O be an Optimal Solution with profit **p'** for the algorithm.
- Let's Suppose that **p'** > **p**. For this to be possible, our algorithm should fail to pick the maximum profit at each comparison that we make. But, our algorithm always picks the local maxima and compares it with global maxima and updates the global maxima to always reflect the maximum possible profit.
- Thus **p'** can never be greater than **p**, a contradiction.

Time Complexity

- Since we need to check m stocks, we run a for loop from 0 to m resulting in O(m).
- And for each stock, we need to check which buy and sell day will result in maximum profit, since there are n days, this gives us O(n).
- Since these loops are nested, we can write O(m) * O(n)

- Therefore, the time complexity is ***O(m * n)***

Space Complexity

- We only need to keep track of the BuyDate, SellDate, MaximumProfit and StockName of the maximum Stock.
- Therefore, the space complexity is constant, i.e. ***O(1)***

2.2.3 Alg 3 - Problem 1 - Dynamic Θ(m ∗ n)

The dynamic algorithm maintains 2 arrays, where one represents the minimum cost for each day and stock and the other represents the maximum profit for each day and stock. It loops through populating these 2 arrays, recording the information of the best transaction for each stock. Then it loops through the list of best transactions, to find the overall optimal one.

$$\mathrm{OPT}(i, j) = \begin{cases} 0 & \text{iff } j = 0 \text{ or } j=1 \\ \max\{\mathrm{OPT}[i][j-1], \underset{\text{for all k in range [0, j]}}{\max}\{\mathrm{price}[i][j] - \mathrm{price}[i][j] + \mathrm{profit}[i-1][j]\})) & \text{iff } j > 1 \end{cases}$$

Correctness

- We use **Proof by Induction** to prove the correctness of this algorithm.
- **Base cases:**
- On **day 1**, our **profit is 0** because if we can only buy and sell the stock on the same day, so we get 0 as profit.
- On **day 2**, our profit will be **max(profit made by not transacting on day 2, profit of selling stock on day two bought on day 1)**. Since these are the only two choices, we know that the maximum of these two values should be the maximum profit we attain on day two, So, this holds true.
- **Induction Step:**
- We will assume that this holds true from **day 3 to day n-1.**
- We need to prove that for a given stock, on day n, we are correctly computing the maximum profit attainable.
- Let's consider the maximum profit attainable on day n.
- Our algorithm considers the maximum of the following two values while computing the maximum profit on day n.
- 1) maximum profit is already obtained by some transaction from **day 1 to day n-1.**
- 2) profit obtained on a transaction by selling the stock on day n, bought during some day between day 1 and day n-1 when the price was minimum.
- From the **induction step,** we know that (1) is true.
- We also know, by **logical reasoning,** that (2) is true, because to maximize profit, we have to buy the stock on the day with the minimum possible value among all days.
- By taking the maximum value of (1) and (2), we will maximize our profit for day n.
- Therefore, **by induction**, we can see that on day n, we get the maximum possible value for some stock x.

- Now, since we are interested in the maximum possible profit of all stocks and the transaction is limited to 1, by simple logical reasoning, we run this algorithm for m stocks and choose the maximum profit among them as the maximum possible profit.

Time Complexity

- We need to consider n days to determine the best buy date and sell date for a stock, which will result in **O(n).**
- Now, we have to do this for m times, since there are m stocks to determine the best possible transaction which will result in **O(m)**.
- Since we check all days for each stock, the loops will be nested, and the time complexity can be written as **O(m) * O(n) = O(m*n).**

Space Complexity

- In this algorithm, we are only storing information about the transaction that produces the maximum possible profit, the space complexity is constant, i.e. O(1).

2.2.4 Alg 4 - Problem 2 - Brute Force $\Theta(m * n^{2k})$

In this brute force algorithm, we find all the possible combinations of non-overlapping transactions (i.e., no two transactions will overlap with each other since a person cannot hold more than two stocks at a time). We use recursion to achieve this. We loop over the number of days and try to find the combination of non-overlapping days which takes one more for loop of size n. Since there are m stocks, we need to use an additional for loop to loop over all the stocks. This function will call itself recursively until it finds the maximum profit for k transactions. Each time, we check the maximum profit and maintain an object where each key is the current maximum profit and the value is the array of transactions to arrive at that value. We return the maximum profit and all the eligible transactions to the user.

Correctness

- We use **Proof by contradiction** to prove the correctness of this algorithm.
- Let's consider that this algorithm outputs maximum profit **p** generated by a **set of k** transactions.
- Now, let's suppose that there is another set of k transactions where profit **p' > p**.
- Our algorithm considered every possible combination of non-overlapping transactions (since we can only hold one share of one stock) at a time.
- This means that our algorithm must have considered the **set of k** transactions producing profit **p'** but found that **p > p'**.
- But, **p'** is assumed to be the maximum, **a contradiction.**

Time Complexity

- The time complexity of this algorithm is O(m*n^2k)
- To make this more clear, consider the following scenario where k = 2.
- Here, our code will run
- n*n*m + (n*n* (m + (n * n * m)))

- Now, if we solve this, we get a time complexity of 2mn^2 + m*n^4.
- Thus, if we consider the largest polynomial, we get O(m*n^4) which would be the case for ***O(m*n^{2k})*** for k=2.

Space Complexity

- The space complexity is ***O(k)*** since we are only interested in maintaining details of k transactions.

2.2.5 Alg 5 - Problem 2 - Dynamic 1 $\Theta(m * n^2 * k)$

In this dynamic programming approach, instead of calculating all non-overlapping transactions, we make a few observations. Instead of recomputing all the non-overlapping transactions k times, we will store the results of each transaction, a subproblem in a 2D array where the row represents the number of transactions and the column represents the span of days it takes to generate profit for the given number of transactions. This will result in a binary choice for us:

1. Either we don't do any transaction on Jth day (because it's not profitable)
2. We do a transaction on the jth day and we calculate the profit as max(price[j] - price[y] + T[i-1][m]) for m from 0 to j

Here, price[j] means the price of the stock on jth day and price[y] means that we consider each day from starting day up until (j-1) day. We also consider the profit of i-1 where i is the current number of transactions done till now. In other words, this means that we check whether current selling a stock on day J that is bought on some previous day (calculated each time) + the profit already generated from i-1 transactions > profit we get by not transacting on that day. We can write this in the below recurrence relation:

```
OPT(k, j) = { 0 if k=0 or j=0
              max( OPT[k][j-1], max(price[j] - price[y] + OPT[k-1][m])   if k>0 and j>0
                                                  for x from day 0 to j-1
```

Correctness

- We use **Proof by induction** to prove the correctness of this algorithm.
- **Base Case**: Consider the maximum profit we can make for **day < 2**. **It's 0**.
- Since we cannot buy anything if no days are given, and if one day is given, we have to buy and sell on the same day resulting in **0 profit**.
- Consider the same case for **k=0 transactions**. If we can't make any transactions, the profit is **0**.
- Now, Another base case is **day = 2 and k=1 transactions** allowed. Now, we calculate this as **max(no transaction on day 2, max(max profit of k-1 transaction upto some day < 2 + profit on buy on day after k-1 transactions and sell on day 2)**.This obviously gives us the correct answer and so, the base case is true.
- **Induction step**: We assume this holds true for days from **3 to n-1** and transactions from **0 to k** across all stocks.

- Now, we need to prove that we obtain the proper value for the **kth transaction** and on **day n** for all stocks.
- To calculate this, our algorithm compares two values.
- The profit we get if **k transactions** are already done before **day n**, and the maximum profit we get by adding the value of previous k-1 transactions up to some day z + we sell some stock x bought before day n, but on or after day z and sold on day n
- We know that **OPT(k,n-1),** the profit of k transactions already done before day n, **holds true** from the **induction step.**
- Now, for calculating the maximum profit on day n, we need the value of the profit available on k-1 transactions up to some day z and the difference between the price on day n and minimum price on or after day z.
- Again, we know the **profit of k-1 transactions is true** up to someday z, is true by the induction step; the profit made by selling on day n + profit of k-1 transactions will be true.
- Therefore, **by induction**, we see that we will have the maximum profit value after k transactions on some day n.

Time Complexity:

- Consider the proposed algorithm
- We run the algorithm k-times to build the DP array O(k) and we run the algorithm n times to build each column of the DP array
- But, while building this, we need to also check the transactions up until jth day, which would require another for loop for 0 to j-1 days
- As an upper bound, we can say that we run two for loops for a total of O(n^2) times
- Then, in each of the kth row and nth column in the DP array, we need to check the maximum of m stocks
- Therefore, we have another for loop to check the best stock and pick it and it runs for m stocks, O(m)
- Since all these for loops are nested, we get O(m) * O(n^2) * O(k) = ***$O(m*k*n^2)$***

Space complexity

- We utilize two 2D arrays to track the profit and transactions. Each array has k+1 rows and n columns where k is the number of transactions allowed and n is the number of days given in the input.
- Therefore, we have a space complexity of
- O(k+1 * n) + O(k+1 * n) = 2 * O(k+1 * n)
- If we simplify this, we get space complexity is ***O((k+1) * n)***

2.2.6 Alg 6 - Problem 2 - Dynamic 2 Θ(m ∗ n * k)

This algorithm is based on the same idea as the algorithm in 2.2.5. Here, we make one key observation to further optimize the calculation of the DP array. Consider the example of filling in the value of k=2 and n=3. In algorithm 2.2.5, we had to obtain the maximum of the following:

i = 0, OPT[1][0] - price[0] + price[3]

i = 1, OPT[1][1] - price[1] + price[3]
I = 2, OPT[1][2] - price[2] + price[3]

Here we see that price[3], or more generally, the price[j] (price on jth day) is common for each iteration. We also see that we only need the max(OPT[1][0] - price[0], OPT[1][1] - price[1], OPT[1][2] - price[2]). Thus, we can modify our recurrence relation as follows:

OPT(k, j) = { 0 if k = 0 or j = 0
max(OPT[k][j-1], price[j] + maxDiff) if k > 0 and j > 0
where maxDiff = max(maxDiff, OPT[k-1][j] - price[j] calculated updated each time for each stock.

Correctness

- This algorithm has the same proof of correctness as the previous algorithm, but we have made one change based on a critical observation
- We only require the maximum possible profit from previous transactions for comparison with calculating the particular current-day transactions.
- We can **prove it intuitively and use method of contradiction**.
- Let's consider how our algorithm calculates and maintains the maximum profit day that is capable of generating the maximum profit.
- For day 1 and k=1, maxDiff is just the price of some stock x on Day 1 - profit so Far for current Day (which is 0 since we can only buy on day 1)).
- For day 2 and k=1 we will store the maxDiff as price on day 1 - profit for k=0 transactions up to day 1, which is just the price of some stock x on day 1.
- For day 3 and k=1, we calculate maxDiff as max(price on day 2 - profit for k=0 transactions up to day2, price on day 1 - profit for k=0 transactions up to day1). Now, we will also update maxDiff based on the values we get for day 3.
- Now, consider some day n, and k=k' transactions, suppose maxDiff there is another value p > maxDiff.
- Our algorithm updates maxDiff each time based on max(maxDiff, profitSofar for current day - price[currentDay]). This means that it encountered p, and determined that p is not the maximum,
- So, p cannot be the maximum which is a **contradiction**.

Time Complexity

- The algorithm loops through *k* times
- It then considers every day through *n*
- Then, it executes for each stock *m*
- Everything else is O(1)
- Since all of these for loops are nested, the time complexity is O(*n*) * O(*m*) * O(*k*), which is ***O(n * m * k)***

Space Complexity

- We utilize two 2D arrays to track the profit and transactions

- Each array has k+1 rows and n columns where k is the number of transactions allowed and n is the number of days given in the input
- In addition, we also utilize two 1D arrays of size m to keep track of the maximum value that can be generated for each m transaction per iteration
- So, the space complexity of this algorithm can be estimated as O(k+1*n) + O(k+1*n) + O(m) + O(m)
- This can be written as 2 O(k+1 * n) + 2 O(m)
- The final space complexity comes down to either O(m) or O(k+1 * n) depending on which is greater.

  ***O(k+1 * n)* if k+1 * n > m**
  ***O(m)* if k+1 * n < m**

2.1.7 Alg 7 - Problem 3 - Brute Force $\Theta(m * 2^n)$

This algorithm considers the problem without the restrictions of k. They can make as many transactions as possible within the c constraint. The brute force method creates a recursive binary tree to compare every combination of days for all of the stocks. This produces an enormous time complexity and is not realistic in practice.

Correctness

- We use **Proof by contradiction** to prove the correctness of this algorithm.
- Our algorithm outputs a set of transactions T, which is capable of producing the maximum possible profit.
- Let's consider that **p** is the maximum profit that is the output of our algorithm.
- Let's assume that there is another value **p',** which is the maximum profit value such that **p' > p.**
- Our algorithm considers all the possible combinations of buy and sells for every Stock taking into account the coolDown period.
- This means that at some point, our algorithm considered the solution **p'** and determined that **p > p'**
- Therefore, this contradicts our assumption that p' > p. Thus **p > p'** and our algorithm is correct.

Time Complexity

- The outer loop executes at most *m* times — once for each stock, which is O(m)
- Then for each stock, for each day, we have two choices, either buy or sell. Since there are n days, the level of the recursion tree will be n. And the total number of possible outcomes we calculate is 2^n. Therefore, the complexity of this operation is **$O(2^n)$**
- Since they are nested, the overall time complexity is ***$O(m * 2^n)$.***

Space Complexity

- We are only interested in tracking the transactions that can produce the maximum output. Thus, at any given time, we can have n-1 transactions at most.
- Therefore, the space complexity is ***O(n - 1)***

2.1.8 Alg 8 - Problem 3 - Dynamic 1 Θ(m $*$ n^2)

This algorithm keeps track of *dp*, which is an array of the maximum profit for each day, and *dpTxns*, which is an array storing the corresponding Transaction objects. It first sets the values for the first two days to determine if a transaction occurs as a base case, and then iterates through the rest. For the rest, it makes sure that transactions are *c* days apart and then determines if the profit would be increased by including a transaction on the current day and current stock. It fills in the values for *dp* and *dpTxns*, where the last value is the maximum. This can then be retraced to get the transactions that contributed to the maximum profit.

```
OPT(i) = { 0 if n=1
        max(OPT(i-1), max((Opt(i-c) or 0) + price[i] - price[j] for j from 0 to i-1
      For n > 1
```

Correctness

- We use **Proof by induction** to prove the correctness of this algorithm.
- **Base case**: Let's consider on day 1, the maximumProfit is 0, because we can buy and sell on the same day so **dp[0] = 0**
- For day 2, we calculate the maximumProfit as max(price[day2] - price[day1], 0) for all stocks.
- Since there are only 2 days, we either make a transaction or we don't make a transaction on day 2. Thus, we store the max of these in **dp[1].**
- **Induction Step**:
- We will assume that this produces maxProfit from day 3 to day n-1 for all stocks and taking c into consideration and store it in dp[3] to dp[n-1] respectively
- We need to prove that we will get the maximum profit on day n for all stocks taking c into consideration.
- Now, for calculating dp[n] we take the maximum of the following values.
    1) We either sell the stock that we are holding from the best possible previous buy day
    2) we Can't sell the stock since we get max Profit from a previous transaction dp[i-1] and we are in a coolDown period.
- From the induction step, we know that (2) holds true.
- From simple logical reasoning, we know that (1) holds true.
- Thus by taking the maximum of these two values, and running for k stocks, we will end up with the maximum possible profit on day n for k stocks.
- By Induction, we see that this algorithm generates the set of transactions that maximizes the profit.

Time Complexity

- Since we are interested in tracking the best days for transacting stocks considering the cool down period, we run the loop from 2 to n days O(n) and for each day up to i, we run a second loop to check for cool down periods O(n), which will result in O(n^2).
- We also need to check for multiple stocks, so we run the loop for O(m) times.
- We also do an initialization step that costs us one O(m) for loop.
- Thus, we can write it as O(m) + O(m)*O(n*n).
- The time complexity is therefore ***O(m * n^2)***

Space Complexity

- This algorithm stores 2 arrays, both of size *n* one for tracking the profits and once for tracking the transactions.
- This can be described as O(2*n*)
- Therefore, we get a space complexity of ***O(n).***

2.1.9 Alg 9 - Problem 3 - Dynamic 2 Θ(m ∗ n)

This algorithm works similar to the brute force algorithm approach we used. Here, we make a key observation that we are calculating the same result over and over again.
For eg, for day 2, we have 2 options, we either sell it, or we buy the stock. We also have to take c into consideration when we are either buying or selling the stocks.
Thus, we end up computing these choices over and over again for larger subProblems.
We instead cache the result and reuse it, reducing redundant computation and slashing the time complexity.

Correctness

- The correctness of this algorithm is **exactly the same as the correctness for the brute force** algorithm for this problem.
- We compute the buy and sell day for each stock and combination taking the coolDown period into consideration.
- The one additional step we are using to optimize the code is Memoizing the sub-problems.

Time Complexity

- We have n days, and for each day, we make a choice either buy or sell. This will result in us calculating a maximum of O(2n) options.
- We have to consider this for m days, so we get O(m)
- By combining these time complexities, we get O(m)*O(2n)
- Therefore, the time complexity is **O(*m* * *n*)**

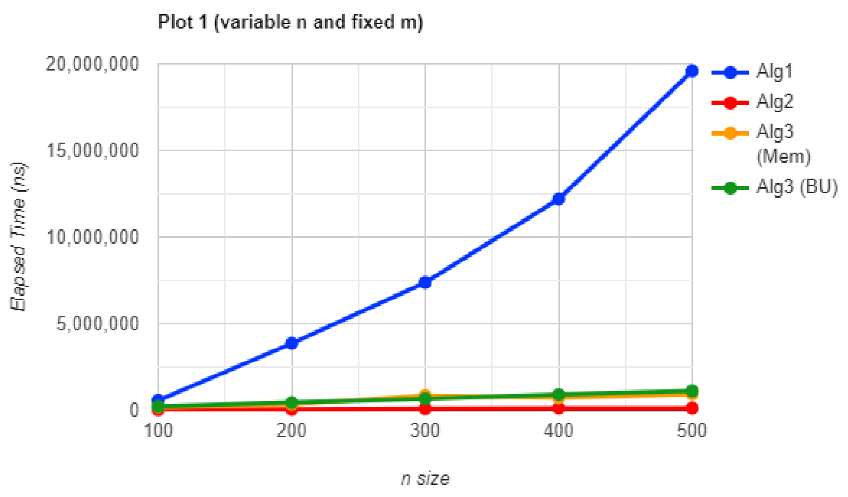
Space Complexity

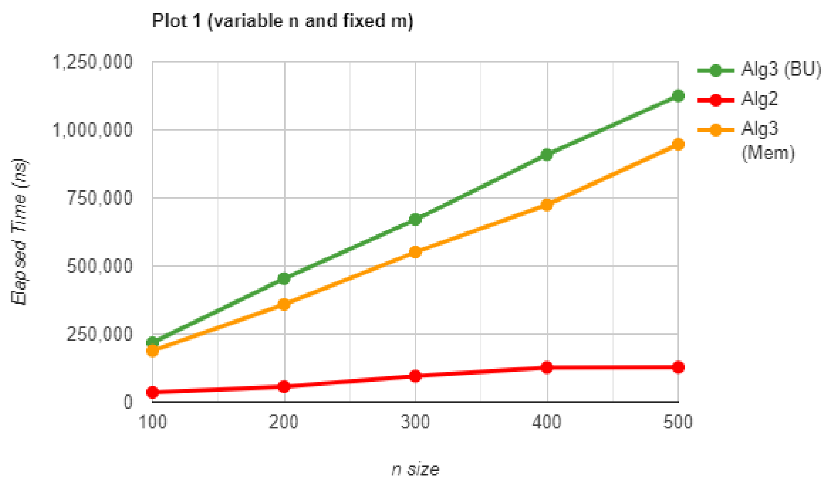
- The algorithm uses a 3D array to store the values of day and stock name along with the choice we make, either buy or sell.
- So, we get a space complexity of size O(2mn)
- Therefore, the space complexity is **O(m*n)**

3. Experimental Comparative Study

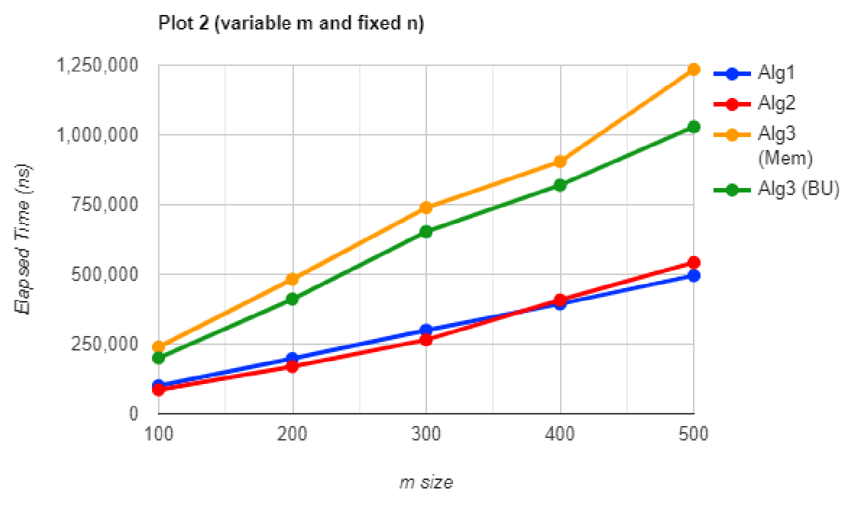
3.1 Problem 1

3.1.1 Variable n and Fixed m



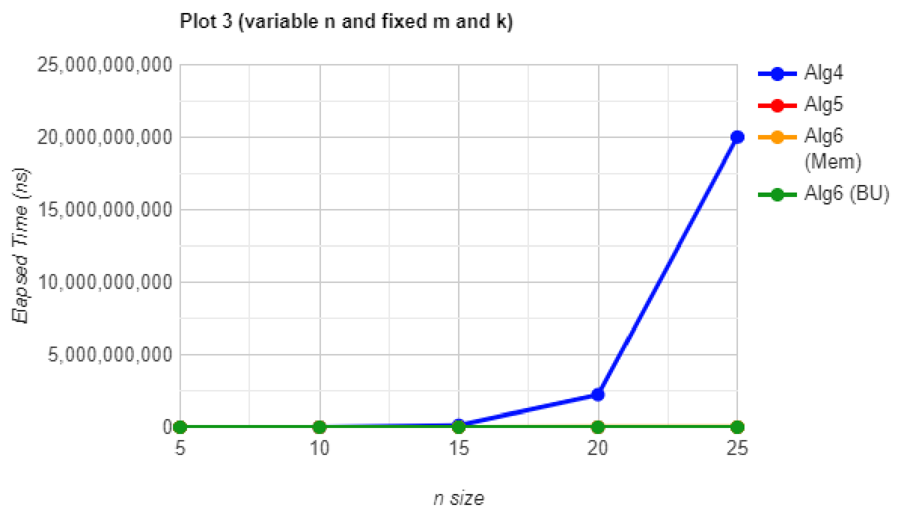


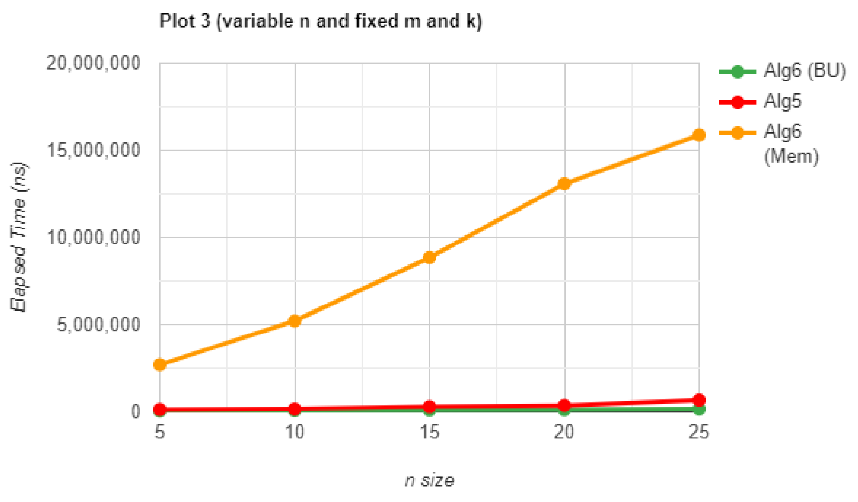
3.1.2 Variable m and Fixed n

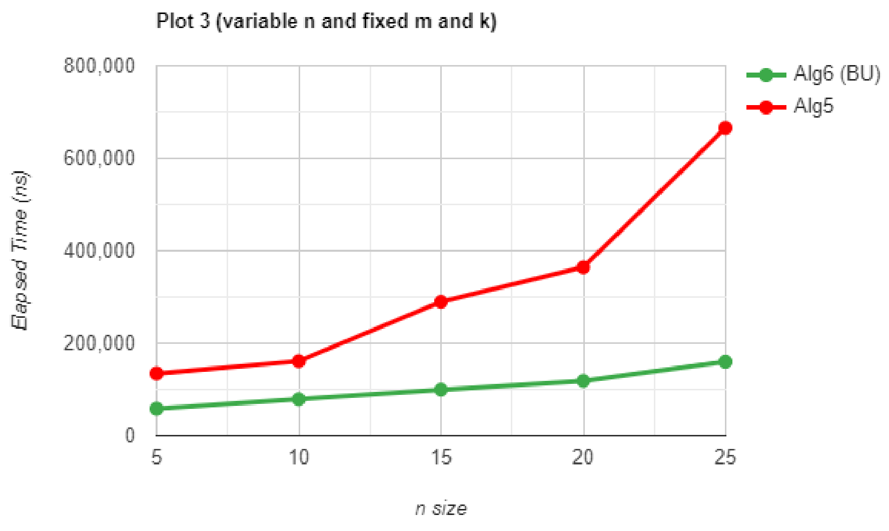


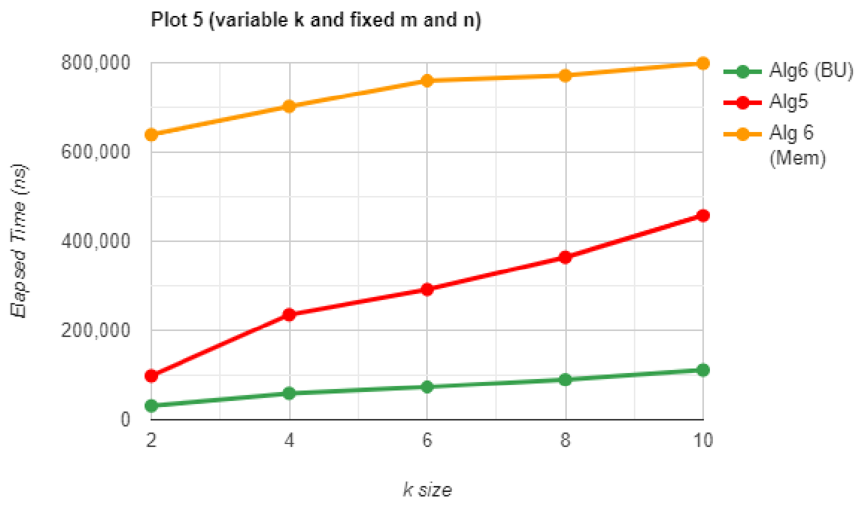
3.2 Problem 2

3.2.1 Variable n and Fixed m and k

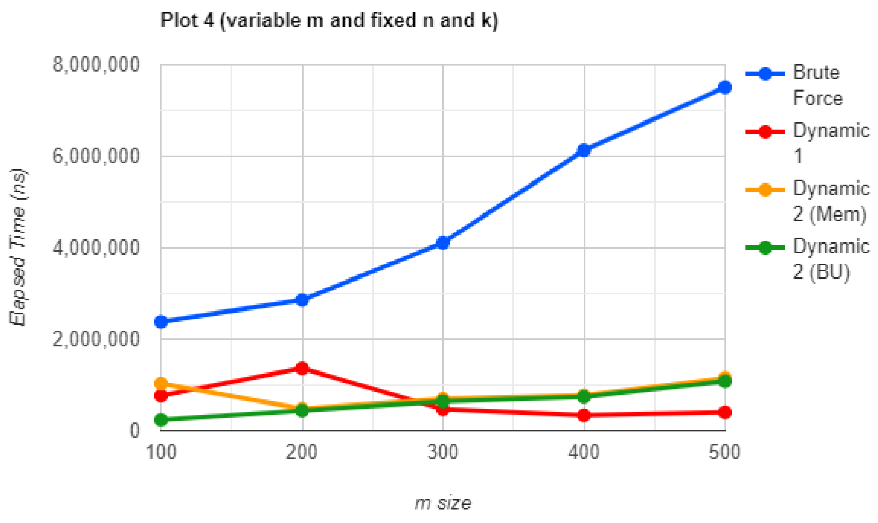



Plot 3 (variable n and fixed m and k)
20,000,000
15,000,000
10,000,000
5,000,000
0
Elapsed Time (ns)
5
10
15
20
25
n size
Alg6 (BU)
Alg5
Alg6 (Mem)

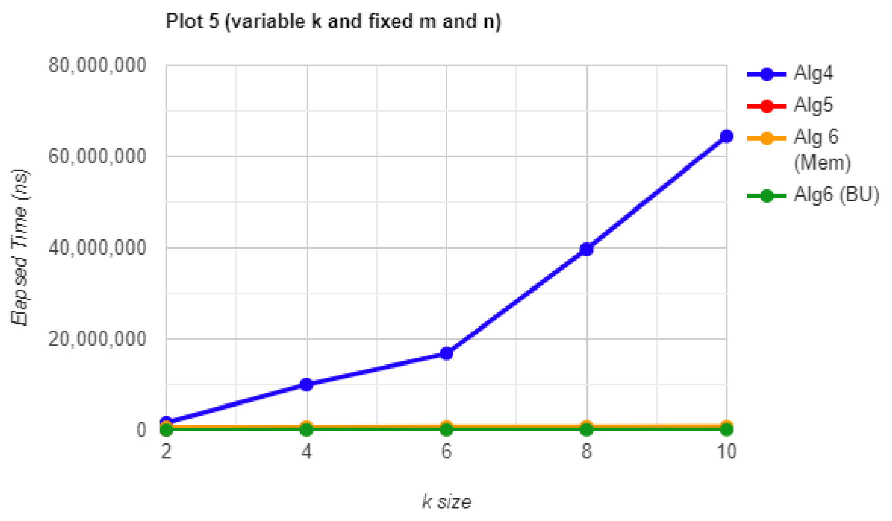

Plot 3 (variable n and fixed m and k)
800,000
600,000
400,000
200,000
0
Elapsed Time (ns)
5
10
15
20
25
n size
Alg6 (BU)
Alg5



3.2.2 Variable m and Fixed n and k

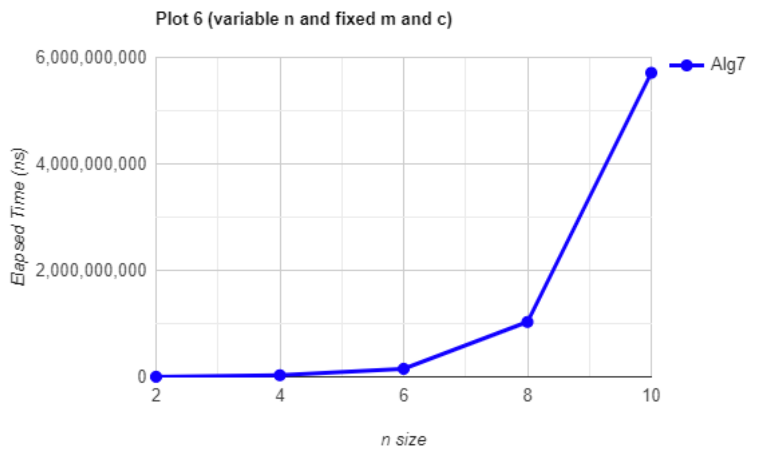


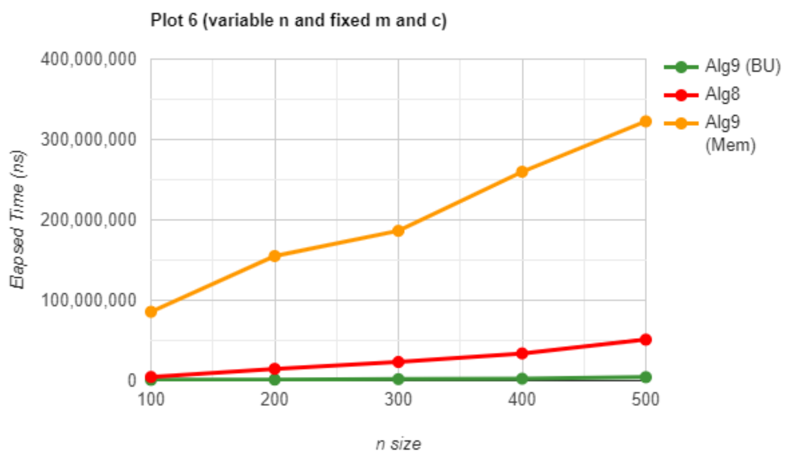
3.2.3 Variable k and Fixed m and n

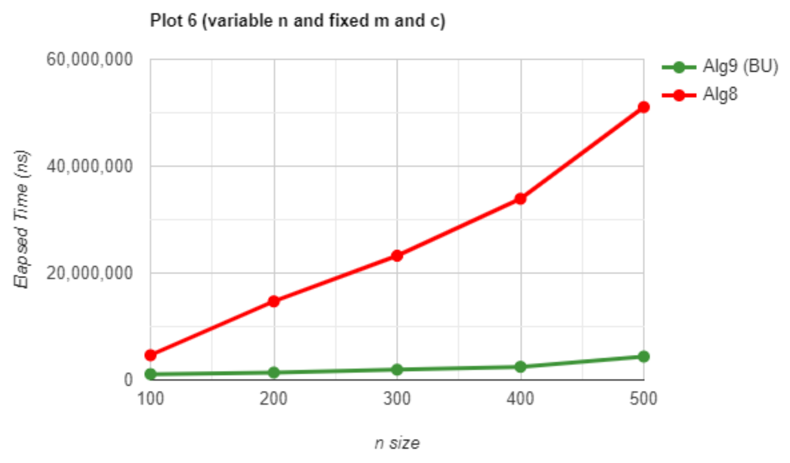


3.3 Problem 3

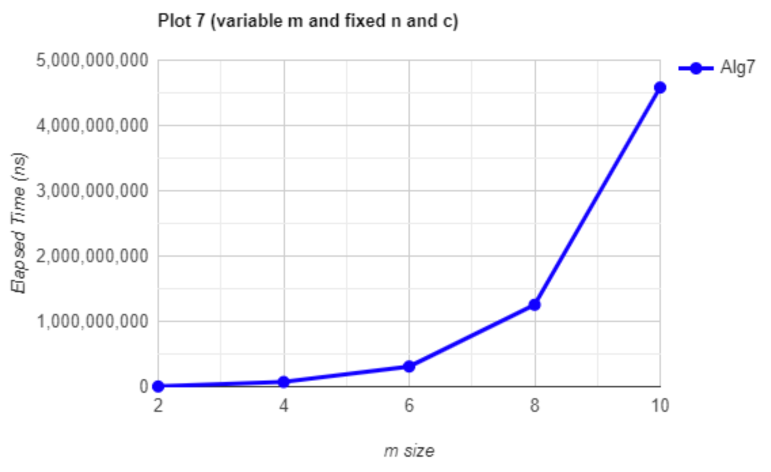
3.3.1 Variable n and Fixed m and c

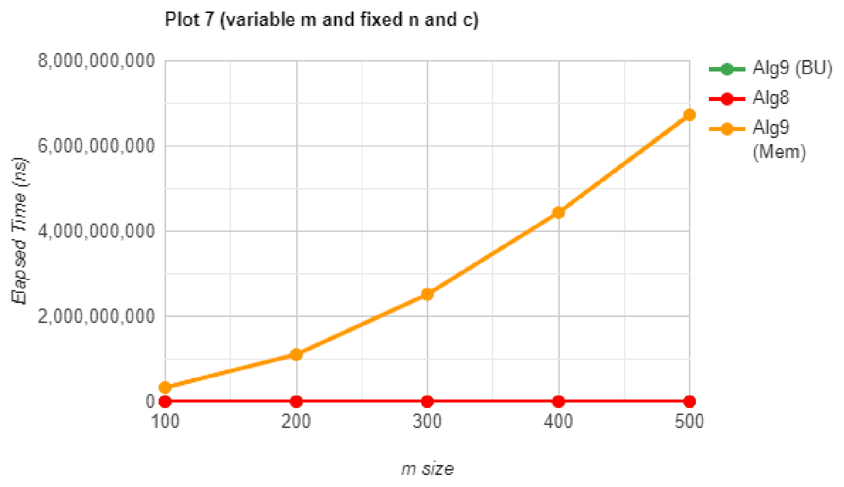


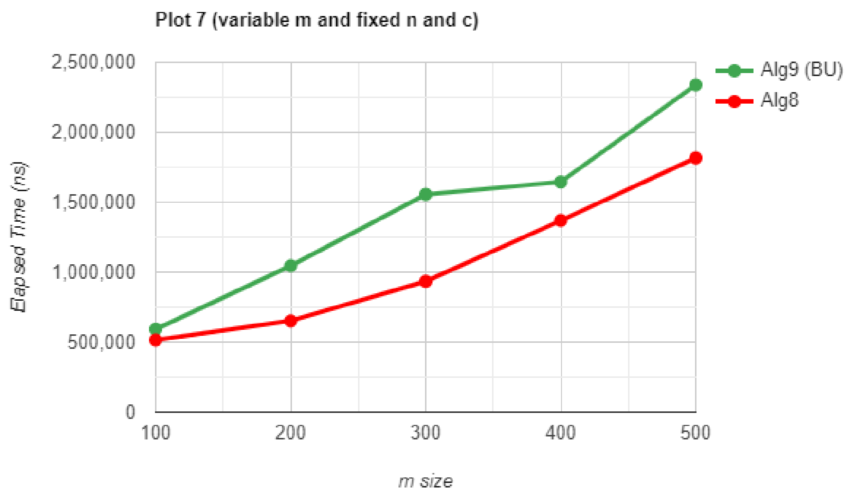




3.3.2 Variable m and Fixed n and c







4. Conclusion

Through this project, we practiced designing, analyzing, and implementing algorithms for a complex problem. We learned how a single problem can be solved via several solutions with varying complexities, and how slight alterations of that problem can change the solution drastically. This project also showed us how the most advanced and complex solution is not always the best solution. For example, with problem 1 the greedy solution had the same time complexity as the dynamic one and performed better during experimentation.

For problem 1, the brute force implementation was relatively straight forward. Once we had the algorithm design, which intuitively consisted of three nested for loops, the implementation flowed from there. Similarly, the greedy and dynamic bottom up implementations flowed nicely after the initial algorithm designs. However, these two did expose some underlying flaws in the algorithm that we later adjusted. The memoization version was the greatest challenge for problem 1. Adjusting the bottom up style algorithm into a recursive dynamic program was not easy, but after looking at examples of how other problems were converted from bottom up to top down, we were able to come up with a solution.

For problem 2, we initially struggled greatly with producing an algorithm design — even for brute force. We spent a very long time discussing and designing before coming up with something that worked. The implementation part went a bit more smoothly, however, we wrote it in Python first as a quick proof of concept. We ran into one major blockage when rewriting it in Java because we forgot that Java uses pass-by-reference. The final implementation in Java for all four versions worked just as good. Finally, we also found it challenging to create the memoized version of the algorithm. This was not as intuitive as with problem one and was less clear than the bottom up version.

Problem 3 was substantially more challenging than the previous two. Without the k variable it made it difficult to determine a base case for our algorithms. We surprisingly struggled more with designing the brute force algorithm than the dynamic ones. We ended up creating Alg8 and Alg9 before it. The implementation flowed easily after the design for all of these because we had gotten the hang of it after the first few implementations, and we used the same test cases from problem 2 with slight modification. In the end, while problem 3 was a much greater challenge it seemed to better represent real world problems.