Class: COP 4533
Professor: Laura Cruz Castro
Term: Fall 2023
Student: Jack Mills
UFID: 3485-7973

Milestone 1

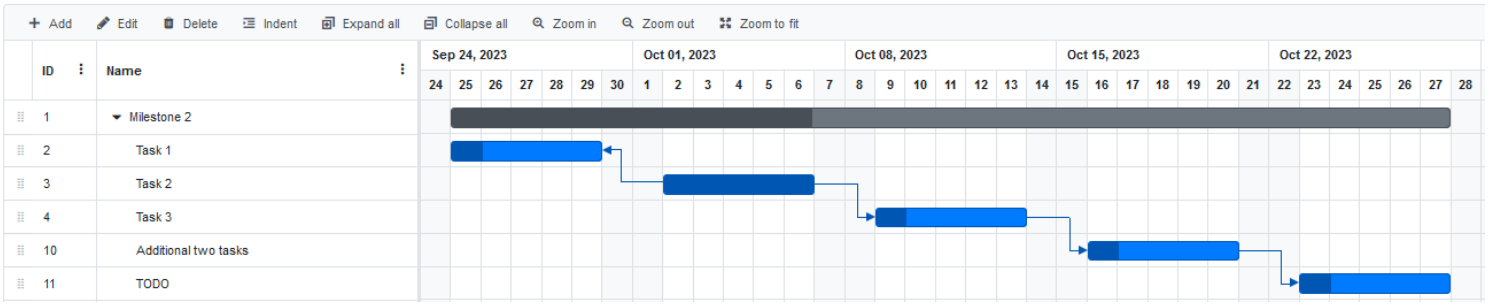
Github Repository Link: https://github.com/jackmillsgator/COP4533FinalProject

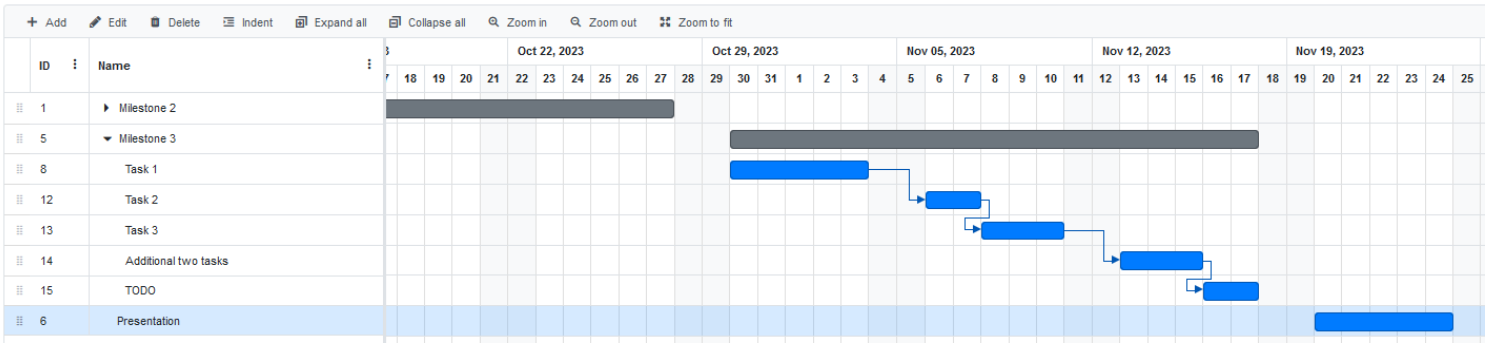
Group Members: Jack Mills UFID3485-7973

Member Roles: I, Jack Mills, am the sole member of my group. So, I'm assuming all Final Project roles.

Communication Method: Not necessary as I'm the only project member.

Project Gantt Chart:





Problem 1.

Steps

1.

Stock	Day 1 Price	Day 2 Price	Day 3 Price	Day 4 Price	Day 5 Price
A	12	1	5	3	16
B	4	4	13	4	9
C	6	8	6	1	2
D	14	3	4	8	10

2.

Stock	Calculation	Buy Day 1, Sell Day 2 - Profit
A	1 - 12 = -11	-11
B	4 - 4 = 0	0
C	8 - 6 = 2	2
D	3 - 13 = -11	-11

Stock	Calculation	Buy Day 2, Sell Day 3 - Profit
A	5 - 1 = 4	4
B	13 - 4 = 9	9
C	6 - 8 = -2	-2
D	4 - 3 = -1	1

Stock	Calculation	Buy Day 3, Sell Day 4 - Profit
A	3 - 5 = -2	-2
B	4 - 13 = -9	-9
C	1 - 6 = -5	-5
D	8 - 4 = 4	4

Stock	Calculation	Buy Day 4, Sell Day 5 - Profit
A	16 - 3 = 13	13
B	9 - 4 = 5	5
C	2 - 1 = 1	1
D	10 - 8 = 2	2

3.

If we are selling the stock the day after purchasing it, then the following are the days with the highest potential profit for each stock:

Stock A: Day 5 with a profit of $13

Stock B: Day 3 with a profit of $9

Stock C: Day 2 with a profit of $2

Stock D: Day 4 with a profit of $4

4.

If we are selling the stock the day after purchasing it, then selling Stock A on Day 5 has the highest potential for profit with a profit of $13.

[(1, 4, 5)]

Problem 2.

Steps

1.

Stock	Day 1 Price	Day 2 Price	Day 3 Price	Day 4 Price	Day 5 Price
A	25	30	15	40	50
B	10	20	30	25	5
C	30	45	35	10	15
D	5	50	35	25	45

2.

As confirmed by Professor Cruz in the Discord, we are assuming that k = 3 (3 transactions) in this problem. And as confirmed by Ayush06 in the Discord, we are also assuming that you can only hold one stock at a time.

There are 2 sequences that yield the same maximum amount of profit:

Sequence 1

		Profit
First transaction:	Buy Stock D on Day 1 and Sell on Day 2	$45
Second Transaction:	Buy Stock B on Day 2 and Sell on Day 3	$10
Third Transaction:	Buy Stock A on Day 3 and Sell on Day 5	$35
Net Profit:		$90

Sequence 2

		Profit
First transaction:	Buy Stock D on Day 1 and Sell on Day 2	$45
Second Transaction:	Buy Stock A on Day 3 and Sell on Day 4	$25
Third Transaction:	Buy Stock D on Day 4 and Sell on Day 5	$20
Net Profit:		$90

3.

[(4, 1, 2), (2, 2, 3), (1, 3, 5)]

AND

[(4, 1, 2), (1, 3, 4), (4, 4, 5)]

Problem 3.

Steps

1.

Stock	Day 1 Price	Day 2 Price	Day 3 Price	Day 4 Price	Day 5 Price	Day 6 Price	Day 7 Price
A	7	1	5	3	6	8	9
B	2	4	3	7	9	1	8
C	5	8	9	1	2	3	10
D	9	3	4	8	7	4	1
E	3	1	5	8	9	6	4

c = 2

2.

Stock	Calculation	Day 1 - Max Profit After c + 1 Days
A	9 - 7 = 2	2
B	9 - 2 = 7	7
C	10 - 5 = 5	5
D	8 - 9 = -1	-1
E	9 - 3 = 6	6

Stock	Calculation	Day 2 - Max Profit After c + 1 Days
A	9 - 1 = 8	8
B	9 - 4 = 5	5
C	10 - 8 = 2	2
D	7 - 3 = 4	4
E	9 - 1 = 8	8

Stock	Calculation	Day 3 - Max Profit After c + 1 Days
A	9 - 5 = 4	4
B	8 - 3 = 5	5
C	10 - 9 = 1	1
D	4 - 4 = 0	0
E	6 - 5 = 1	1

Stock	Calculation	Day 4 - Max Profit After c + 1 Days
A	9 - 3 = 6	6
B	8 - 7 = 1	1
C	10 - 1 = 9	9
D	1 - 8 = -7	-7
E	4 - 8 = -4	-4

3.

There are 5 sequences that yields the same maximum amount of profit:

Sequence 1

		Profit
First transaction:	Buy Stock C on Day 1 and sell on Day 2	$3
	Wait till Day 5 (i = 2 so i + c + 1 = 2 + 2 + 1 = 5 as in waiting till Day 5)	
Second transaction:	Buy Stock C on Day 5 and sell on Day 7	$8
Net profit:		$11

Sequence 2

First transaction:	Buy Stock A on Day 2 and sell on Day 3	$4
	Wait till Day 5 (i = 3 so i + c + 1 = 3 + 2 + 1 = 6 as in waiting till Day 6)	
Second transaction:	Buy Stock B on Day 6 and sell on Day 7	$7
Net profit:		$11

Sequence 3

First transaction:	Buy Stock A on Day 2 and sell on Day 3	$4
	Wait till Day 5 (i = 3 so i + c + 1 = 3 + 2 + 1 = 6 as in waiting till Day 6)	
Second transaction:	Buy Stock C on Day 6 and sell on Day 7	$7
Net profit:		$11

Sequence 4

First transaction:	Buy Stock E on Day 2 and sell on Day 3	$4
	Wait till Day 5 (i = 3 so i + c + 1 = 3 + 2 + 1 = 6 as in waiting till Day 6)	
Second transaction:	Buy Stock B on Day 6 and sell on Day 7	$7
Net profit:		$11

Sequence 5

First transaction:	Buy Stock E on Day 2 and sell on Day 3	$4
	Wait till Day 5 (i = 3 so i + c + 1 = 3 + 2 + 1 = 6 as in waiting till Day 6)	
Second transaction:	Buy Stock C on Day 6 and sell on Day 7	$7
Net profit:	$11	

3.

[(3, 1, 2), (3, 5, 7)]

AND

[(1, 2, 3), (2, 6, 7)]

AND

[(1, 2, 3), (3, 6, 7)]

AND

[(5, 2, 3), (2, 6, 7)]

AND

[(5, 2, 3), (3, 6, 7)]

Milestone 2

C++ will be used to implement each of the algorithms

Task - 1

Design a brute force algorithm for solving Problem 1 that runs in O(m · n^2) time.

Example Table

Stock	Day 1 Price	Day 2 Price	Day 3 Price	Day 4 Price	Day 5 Price
A	12	1	5	3	16
B	4	4	13	4	9
C	6	8	6	1	2
D	14	3	4	8	10

With the given time complexity of O(m · n^2), we know that n^2 could describe a nested for loop that will search through each element of a 2d array. And because there is also ‘m’ within the time complexity, it appears that we have an additional, novel loop through our data.

Algorithm

// We assume that this array is already filled with the price data (like in the table above).
array predictedPrices[]

// We want to initialize maxProfit with a large negative number since, in theory, the maximum
// amount of profit could be a negative number if you can only lose money by selling any of the
// stocks in the future.
int variable maxProfit = -999999

// In addition to storing the value of the maximum profit made, we want to store the buy and sell
// days associated with the max profit so we know which indices within the 2d array to reference.
int maxProfitBuyDay = 0
int maxProfitSellDay = 0

Looping through each row in the predictedPrices 2d array.

> Looping through each element in every row of the predictedPrices 2d array.
>
> > Find the profit by subtracting each previous index value from the current index value. For example, if we're currently evaluating Day 4 of Stock A (looking at my example table above), we find each possible profit selling that day by subtracting the Day 1 price from the Day 4 price, the Day 2 price from the the Day 4 price, and the Day 3 price from the Day 4 price.
> >
> > We keep track of the highest possible profit by comparing each calculated profit between two days of the same stock with the current value of the maxProfit variable. If a calculated profit happens to be larger (higher profit) than the current value of the maxProfit variable, the calculated profit becomes the new value of maxProfit.

Print or use the maxProfit value since it is the desired transaction that yields the maximum amount of profit.

Task - 2

Design a greedy algorithm for solving Problem 1 that runs in O(m · n) time.

Example Table

Stock	Day 1 Price	Day 2 Price	Day 3 Price	Day 4 Price	Day 5 Price
A	12	1	5	3	16
B	4	4	13	4	9
C	6	8	6	1	2
D	14	3	4	8	10

Because we're solving the same problem as the problem in Task 1 while using a Greedy algorithm instead, I would expect this algorithm to be similar to Task 1's algorithm but, however, contain certain local optimizations at the expense of optimization for the entire problem to be solved as a whole. And because the time complexity is O(m · n), I would believe that some of the local optimization would lead to an absence of nested for loops in the same manner as Task 1's algorithm (the time complexity does not contain n^2 like that of Task 1's time complexity). Although the algorithm below has a significant chance of not finding the correct transaction that will yield the maximum profit, it makes greedy current (present-time) optimizations at the expense of the overall optimization, and in doing so, it avoids a time complexity containing n^2 because the current optimization avoids a nested loop that iterates through all 2d array indices.

Algorithm

// We assume that this array is already filled with the price data (like in the table above).
array predictedPrices[]

// We want to initialize maxProfit with a large negative number since, in theory, the maximum
// amount of profit could be a negative number if you can only lose money by selling any of the
// stocks in the future.
int variable maxProfit = -999999

// In addition to storing the value of the maximum profit made, we want to store the buy and sell
// days associated with the max profit so we know which indices within the 2d array to reference.
int maxProfitBuyDay
int maxProfitSellDay

// This stores the row with the smallest value in its index[0].
int smallestIndexZeroRow

Looping through each row in the predictedPrices 2d array.

> Find the index[0] with the smallest value for a buy price (in this case it will be row[1] with a buy price for stock B of $4).

Looping through each element in the row with the smallest buy value in rows index[0].

> Find the profit by subtracting each previous index value from the current index value within the row. For example, if Stock B (looking at my example table above) is chosen as having the smallest value for its index[0], and we're currently evaluating Day 4 of Stock B, we find each possible profit selling that day by subtracting the Day 1 price from the Day 4 price, the Day 2 price from the Day 4 price, and the Day 3 price from the Day 4 price.
>
> We keep track of the highest possible profit by comparing each calculated profit between two days of the same stock with the current value of the maxProfit variable. If a calculated profit happens to be larger (higher profit) than the current value of the maxProfit variable, the calculated profit becomes the new value of maxProfit.

Print or use the maxProfit value since it is the desired transaction that yields the maximum amount of profit.

Task - 3

Design a dynamic programming algorithm for solving Problem 1 that runs in $O(m \cdot n)$ time.

Example Table

Stock	Day 1 Price	Day 2 Price	Day 3 Price	Day 4 Price	Day 5 Price
A	12	1	5	3	16
B	4	4	13	4	9
C	6	8	6	1	2
D	14	3	4	8	10

Because we're solving the same problem as the problem in Task 2 (including a lack of n^2 time complexity) while using a dynamic algorithm instead, I would expect the resulting algorithm to be similar to Task 2's algorithm but, however, contain memoization which is the storing of results to subproblems so that they are immediately available without the need for recalculation. This technique of memoization is often implemented in cases where similar calculations are repeatedly made like in the case of recursion. Because we are repeatedly doing similar calculations when trying to find the maximum price by subtracting previous stock prices from a later (future) stock price, we want to implement memoization here.

Algorithm

// We assume that this array is already filled with the price data (like in the table above).
array predictedPrices[]

// We want to initialize maxProfit with a large negative number since, in theory, the maximum
// amount of profit could be a negative number if you can only lose money by selling any of the
// stocks in the future.
int variable maxProfit = -999999

// In addition to storing the value of the maximum profit made, we want to store the buy and sell
// days associated with the max profit so we know which indices within the 2d array to reference.
int maxProfitBuyDay
int maxProfitSellDay

// This stores the row with the smallest value in its index[0].
int smallestIndexZeroRow

Using recursive calls to iterate through each row in the predictedPrices 2d array.

> // To find the max profit for each row, we want to use recursion to calculate the difference
> // in price for each current element and previous element pair that is possible (all
> // possible future and past stock price pairs that could yield a profit). During each
> // recursive call, we can compare the current profit value and the returned profit value
> // from a recursive call, and we would return the larger of the two. We would do this
> // until the final returned value is returned and stored into maxProfit.
>
> If we have not reached the last row in the 2d array predictedPrices, utilize recursion to find maxProfit of the next row.
>
> > Use recursion to return the maxProfit from each row.
>
> The value returned from each recursive call to the following row is then compared to the current value of maxProfit until the largest of the values is stored in maxProfit and returned.

Print or use the maxProfit value since it is the desired transaction that yields the maximum amount of profit.

Task - 6

Design a dynamic programming algorithm for solving Problem 3 that runs in O(m · n^2) time.

Example Table and Value for c

Stock	Day 1 Price	Day 2 Price	Day 3 Price	Day 4 Price	Day 5 Price	Day 6 Price	Day 7 Price
A	7	1	5	3	6	8	9
B	2	4	3	7	9	1	8
C	5	8	9	1	2	3	10
D	9	3	4	8	7	4	1
E	3	1	5	8	9	6	4

c = 2

Because our time complexity contains n^2, we would expect that our algorithm would contain a nested for loop. Therefore, we're going to iterate through all elements of the 2d array containing the stock prices by using a nested for loop, and because we're utilizing dynamic programming, we're going to use recursion to return the maximum profit selling a stock for each day by subtracting previous stock prices from the current stock price, comparing the current value of maxProfit with the profit value returned from each recursion call, returning the larger of the two values each time, and ultimately returning the max profit possible for selling each stock on a particular day.

Algorithm

// We assume that this array is already filled with the price data (like in the table above).
array predictedPrices[]

// We want to initialize maxProfit with a large negative number since, in theory, the maximum
// amount of profit could be a negative number if you can only lose money by selling any of the
// stocks in the future.
int variable maxProfit = -999999

// In addition to storing the value of the maximum profit made, we want to store the buy and sell
// days associated with the max profit so we know which indices within the 2d array to reference.
int maxProfitBuyDay
int maxProfitSellDay

// This stores the row with the smallest value in its index[0].
int smallestIndexZeroRow

Using recursive calls to iterate through each row in the predictedPrices 2d array.

Looping through each element in every row of the predictedPrices 2d array.

// To find the max profit for each stock on each day, we want to use recursion to
// calculate the difference in price between the current price and every previous
// price. During each recursive call, we can compare the current profit value and
// the returned profit value from a recursive call. Then we would return the larger
// of the two. We would do this until the final return value for each row is stored
// into a 2d array.

We would store the information for the maximum profit for each row (stock) into a 2d array called maxProfitInfo. For each row in this 2d array, we would store the following: the buy day, the sell day, the maxProfit.

We would then calculate the sequence of buys and sells that yield the highest amount of profit while taking into consideration the c value for the waiting period.

Print or use the desired sequence of transactions that yield the maximum profit.

Task - 7

Design a dynamic programming algorithm for solving Problem 3 that runs in $O(m \cdot n)$ time.

Example Table and Value for c

Stock	Day 1 Price	Day 2 Price	Day 3 Price	Day 4 Price	Day 5 Price	Day 6 Price	Day 7 Price
A	7	1	5	3	6	8	9
B	2	4	3	7	9	1	8
C	5	8	9	1	2	3	10
D	9	3	4	8	7	4	1
E	3	1	5	8	9	6	4

c = 2

This algorithm will be almost exactly the same as the algorithm created in Task 6. However, with a time complexity of $O(m \cdot n)$, we are going to replace the nested for loop with recursive calls within each row for each possible pair of a future and previous stock price that could yield a profit.

Algorithm

// We assume that this array is already filled with the price data (like in the table above).
array predictedPrices[]

// We want to initialize maxProfit with a large negative number since, in theory, the maximum
// amount of profit could be a negative number if you can only lose money by selling any of the
// stocks in the future.
int variable maxProfit = -999999

// In addition to storing the value of the maximum profit made, we want to store the buy and sell
// days associated with the max profit so we know which indices within the 2d array to reference.
int maxProfitBuyDay
int maxProfitSellDay

// This stores the row with the smallest value in its index[0].
int smallestIndexZeroRow

Using recursive calls to iterate through each row in the predictedPrices 2d array.

> // To find the max profit for each row, we want to use recursion to calculate the difference
> // in price for each current element and previous element pair that is possible (all
> // possible future and past stock price pairs that could yield a profit). During each
> // recursive call, we can compare the current profit value and the returned profit value
> // from a recursive call. Then we would return the larger of the two. We would do this
> // until the final return value is stored into maxProfit.
>
> If we have not reached the last row in the 2d array predictedPrices, utilize recursion to find maxProfit of the next row.
>
> > Use recursion to return the maxProfit from each row.
>
> We would store the information for the maximum profit for each row (stock) into a 2d array called maxProfitInfo. For each row in this 2d array, would would store the following: the buy day, the sell day, the maxProfit.

We would then calculate the sequence of buys and sells that yield the highest amount of profit while taking into consideration the c value for the waiting period.

Print or use the desired sequence of transactions that yield the maximum profit.

Milestone 3

Task - 1

Algorithm Implementation

```
#include <iostream>
#include <vector>
#include <string>

using namespace std;

vector<int> findMaxProfit(vector<vector<int>> predictedPrices);
void printMaxProfitValues(vector<int> maxProfitValues, string vectorName);

int main()
{

    //  I am declaring and initializing the 2D vectors to test the
    //  implementation of the algorithm.

    ////////////////////////////////////////////////////////////////////
    //                                                                //
    //                 STOCK       BUY DAY     SELL DAY    PROFIT     //
    //      ANSWER:    1,          2,          5,          15         //
    //                                                                //
    ////////////////////////////////////////////////////////////////////
    vector<vector<int>> predictedPrices1
    {

        {12, 1, 5, 3, 16},
        {4, 4, 13, 4, 9},
        {6, 8, 6, 1, 2},
        {14, 3, 4, 8, 10}

    };
```

```
////////////////////////////////////////////////////////////////////
//                                                                //
//             STOCK       BUY DAY     SELL DAY    PROFIT         //
//  ANSWER:    4,          2,          3,          11             //
//                                                                //
////////////////////////////////////////////////////////////////////
vector<vector<int>> predictedPrices2
{

    {7, 1, 5, 3, 6},
    {2, 4, 3, 7, 9},
    {5, 8, 9, 1, 2},
    {9, 3, 14, 8, 7}

};

////////////////////////////////////////////////////////////////////
//                                                                //
//             STOCK       BUY DAY     SELL DAY    PROFIT         //
//  ANSWER:    0,          0,          0,          0              //
//                                                                //
////////////////////////////////////////////////////////////////////
vector<vector<int>> predictedPrices3
{

    {12, 8, 5, 3, 1},
    {20, 16, 14, 8, 7},
    {5, 4, 3, 2, 1},
    {0, 0, 0, 0, 0}

};
```

```
    //  I'm calling the findMaxProfit() function, which is the implementation
    //  of the algorithm, and the printMaxProfitValues() function that will
    //  confirm that the correct values are returned.
    vector<int> maxProfitValues1 = findMaxProfit(predictedPrices1);
    printMaxProfitValues(maxProfitValues1, "predictedPrices1");

    vector<int> maxProfitValues2 = findMaxProfit(predictedPrices2);
    printMaxProfitValues(maxProfitValues2, "predictedPrices2");

    vector<int> maxProfitValues3 = findMaxProfit(predictedPrices3);
    printMaxProfitValues(maxProfitValues3, "predictedPrices3");

    return 0;
}

//  Implementation of the algorithm.
vector<int> findMaxProfit(vector<vector<int>> predictedPrices)
{

    //  I'm declaring and initializing all necessary variables.
    long unsigned int i = 0;
    long unsigned int j = 0;
    long unsigned int k = 0;

    int maxProfitStock = 0;
    int maxProfitBuyDay = 0;
    int maxProfitSellDay = 0;
    int maxProfit = 0;
    int tempInt = 0;

    //  I'm creating a vector of size 4 filled with 0 as all of its values.
    //  If a max profit from a transaction is, indeed, found, then the 0's
    //  will change to other values.
    vector<int> maxProfitValues(4, 0);
```

```
//  ************************************************************  //
//                                                               //
//  With the given time complexity of O(m · n^2), we know that n^2  //
//  could describe a nested for loop that will search through each  //
//  element of a 2d array. And because there is also 'm' within the  //
//  time complexity, it appears that we have an additional, novel  //
//  loop through our data.                                       //
//                                                               //
//  ************************************************************  //

//  Iterating through each stock
for (i = 0; i < predictedPrices.size(); i++)
{

    //  Iterating through the prices for each stock
    for (j = 0; j < predictedPrices[i].size(); j++)
    {

        //  I'm going through each previous price before the current
        //  stock price (price at index j) and subtracting each
        //  previous stock from the current stock. If the sum of
        //  that operation is larger than the current value of
        //  maxProfit, then the sum becomes the new value of
        //  maxProfit.
        for (k = 0; k < j; k++)
        {

            tempInt = predictedPrices[i][j] - predictedPrices[i][k];
```

```
            //  If a new, larger value for maxProfit is found, we
            //  want to store and keep track of the new value for
            //  value for maxProfit as well as other information
            //  regarding that transaction for when we print the
            //  desired tuple of informational values.
            if (tempInt > maxProfit)
            {
                maxProfitStock = i;
                maxProfitBuyDay = k;
                maxProfitSellDay = j;
                maxProfit = tempInt;
            }

        }

    }

}

//  If we, indeed, found a max profit of some kind, then we want
//  to add 1 to the values representing the stock row, buy day,
//  and sell day. The reason is because the format given in the
//  Final Project instructions gives examples where tuples
//  contain values of the actual row and column number as opposed
//  to programming index values (values that start with 0 i.e.
//  0, 1, 2, 3, ...).
if (maxProfit > 0)
{
    maxProfitValues[0] = maxProfitStock + 1;
    maxProfitValues[1] = maxProfitBuyDay + 1;
    maxProfitValues[2] = maxProfitSellDay + 1;
    maxProfitValues[3] = maxProfit;
}
```

```
    //  If maxProfit is 0, then no transaction was found that would
    //  lead to a profit. So we want to return the tuple (0, 0, 0, 0).
    else if (maxProfit <= 0)
    {
        maxProfitValues[0] = maxProfitStock;
        maxProfitValues[1] = maxProfitBuyDay;
        maxProfitValues[2] = maxProfitSellDay;
        maxProfitValues[3] = maxProfit;
    }

    return maxProfitValues;

}

//  Simply prints values to see if findMaxProfit() is working correctly
void printMaxProfitValues(vector<int> maxProfitValues, string vectorName)
{

    cout << endl;
    cout << "---------------------------------" << endl;
    cout << "\t" << vectorName << endl;
    cout << "---------------------------------" << endl;

    cout << endl;
    cout << "Tuple Output: " << endl;
    cout << endl;
    cout << "(" << maxProfitValues[0]   << ", "
                << maxProfitValues[1]   << ", "
                << maxProfitValues[2]   << ", "
                << maxProfitValues[3]   << ")";
    cout << endl;
    cout << endl;

}
```

Test Cases

```
////////////////////////////////////////////////////////////////////
//                                                                //
//                 STOCK       BUY DAY     SELL DAY    PROFIT     //
//      ANSWER:    1,          2,          5,          15         //
//                                                                //
////////////////////////////////////////////////////////////////////
vector<vector<int>> predictedPrices1
{

    {12, 1, 5, 3, 16},
    {4, 4, 13, 4, 9},
    {6, 8, 6, 1, 2},
    {14, 3, 4, 8, 10}

};

////////////////////////////////////////////////////////////////////
//                                                                //
//              STOCK       BUY DAY     SELL DAY    PROFIT        //
//  ANSWER:     4,          2,          3,          11            //
//                                                                //
////////////////////////////////////////////////////////////////////
vector<vector<int>> predictedPrices2
{

    {7, 1, 5, 3, 6},
    {2, 4, 3, 7, 9},
    {5, 8, 9, 1, 2},
    {9, 3, 14, 8, 7}

};

////////////////////////////////////////////////////////////////////
//                                                                //
//              STOCK       BUY DAY     SELL DAY    PROFIT        //
//  ANSWER:     0,          0,          0,          0             //
//                                                                //
////////////////////////////////////////////////////////////////////
vector<vector<int>> predictedPrices3
{

    {12, 8, 5, 3, 1},
    {20, 16, 14, 8, 7},
    {5, 4, 3, 2, 1},
    {0, 0, 0, 0, 0}

};
```

Test Results

```
PS C:\Users\jmmil\OneDrive\Desktop\Milestone3\Task1> mingw32-make
g++ -c -Wall -std=c++14 main.cpp
g++ main.o -o task1
PS C:\Users\jmmil\OneDrive\Desktop\Milestone3\Task1> mingw32-make run
./task1

---------------------------------
        predictedPrices1
---------------------------------

Tuple Output:

(1, 2, 5, 15)


---------------------------------
        predictedPrices2
---------------------------------

Tuple Output:

(4, 2, 3, 11)


---------------------------------
        predictedPrices3
---------------------------------

Tuple Output:

(0, 0, 0, 0)

PS C:\Users\jmmil\OneDrive\Desktop\Milestone3\Task1>
```

Description of the Implementation of the Algorithm:

Because of the n^2 found within the O(m * n^2) time complexity, I implemented a nested for loop that iterates through all elements found within the 2d array that contained the prices. Then, accounting for the m found within the given time complexity, I iterated through all prices that came before the current price within each stock row in order to subtract that previous price. Throughout all of these operations, I saved the difference operation that yielded the largest value as well as other information regarding the transaction that yielded that largest value in order to output the wanted tuple.

Limitations and Trade-offs:

Pros

- Creating an algorithm using this approach is comparatively quite simple and straight-forward.
- Because these algorithms utilize techniques like deeply nested for loops that visit each and every element in order to solve the problem at hand, they almost ensure a correct solution if the algorithm is written correctly.
- Great for tackling small problems for problems with little elements at play due to their simplicity.

Cons

- These algorithms can be very slow. You can't use brute force when time efficiency is important.
- The inefficiency of brute force algorithms can lead to a dependency upon a machine's ability to compute in order to get efficient results. Instead, algorithms produced that maximize time efficiency can eliminate this and make results and responsiveness quicker and practical.

Comparative Analysis:

Compared to my Task 2 and Task 3 algorithm implementations of Problem 1, a brute force approach will be simpler to implement but far less time efficient.

Analysis of the Algorithm and Lessons Learned:

Brute force algorithms have their place in solving simple problems with little variables at play. Some problems don't require the absolutely most time efficient algorithm. However, when the number of computations begin to increase, a brute force approach quickly shows as a slow, inefficient means to find an answer

Task - 2

Algorithm Implementation

```
#include <iostream>
#include <vector>
#include <string>

using namespace std;

vector<int> findMaxProfit(vector<vector<int>> predictedPrices);
void printMaxProfitValues(vector<int> maxProfitValues, string vectorName);

int main()
{

    //  I am declaring and initializing the 2D vectors to test the
    //  implementation of the algorithm.

    ////////////////////////////////////////////////////////////////////
    //                                                                //
    //                  STOCK       BUY DAY     SELL DAY    PROFIT    //
    //      ANSWER:     1,          2,          5,          15        //
    //                                                                //
    ////////////////////////////////////////////////////////////////////
    vector<vector<int>> predictedPrices1
    {

        {12, 1, 5, 3, 16},
        {4, 4, 13, 4, 9},
        {6, 8, 6, 1, 2},
        {14, 3, 4, 8, 10}

    };
```

```
////////////////////////////////////////////////////////////////////////
//                                                                    //
//              STOCK       BUY DAY     SELL DAY    PROFIT            //
//  ANSWER:     4,          2,          3,          11                //
//                                                                    //
////////////////////////////////////////////////////////////////////////
vector<vector<int>> predictedPrices2
{

    {7, 1, 5, 3, 6},
    {2, 4, 3, 7, 9},
    {5, 8, 9, 1, 2},
    {9, 3, 14, 8, 7}

};

////////////////////////////////////////////////////////////////////////
//                                                                    //
//              STOCK       BUY DAY     SELL DAY    PROFIT            //
//  ANSWER:     0,          0,          0,          0                 //
//                                                                    //
////////////////////////////////////////////////////////////////////////
vector<vector<int>> predictedPrices3
{

    {12, 8, 5, 3, 1},
    {20, 16, 14, 8, 7},
    {5, 4, 3, 2, 1},
    {0, 0, 0, 0, 0}

};

//  I'm calling the findMaxProfit() function, which is the implementation
//  of the algorithm, and the printMaxProfitValues() function that will
//  confirm that the correct values are returned.
vector<int> maxProfitValues1 = findMaxProfit(predictedPrices1);
printMaxProfitValues(maxProfitValues1, "predictedPrices1");
```

```
    vector<int> maxProfitValues2 = findMaxProfit(predictedPrices2);
    printMaxProfitValues(maxProfitValues2, "predictedPrices2");

    vector<int> maxProfitValues3 = findMaxProfit(predictedPrices3);
    printMaxProfitValues(maxProfitValues3, "predictedPrices3");

    return 0;

}

//  Implementation of the algorithm.
vector<int> findMaxProfit(vector<vector<int>> predictedPrices)
{

    //  I'm declaring and initializing all necessary variables.
    long unsigned int i = 0;
    long unsigned int j = 0;
    long unsigned int k = 0;

    int maxProfitStock = 0;
    int maxProfitBuyDay = 0;
    int maxProfitSellDay = 0;
    int maxProfit = 0;
    int tempInt = 0;

    int smallestIndexZeroVALUE = 999999;
    int smallestIndexZeroROW = 0;

    //  I'm creating a vector of size 4 filled with 0 as all of its values.
    //  If a max profit from a transaction is, indeed, found, then the 0's
    //  will change to other values.
    vector<int> maxProfitValues(4, 0);
```

```
//  ************************************************************  //
//                                                               //
//  Because we're solving the same problem as the problem in Task 1  //
//  while using a Greedy algorithm instead, I would expect this      //
//  algorithm to be similar to Task 1's algorithm but, however,      //
//  contain certain local optimizations at the expense of           //
//  optimization for the entire problem to be solved as a whole. And //
//  because the time complexity is O(m · n), I would believe that    //
//  some of the local optimization would lead to an absence of nested //
//  for loops in the same manner as Task 1's algorithm (the time     //
//  complexity does not contain n^2 like that of Task 1's time       //
//  complexity). Although the algorithm below has a significant      //
//  chance of not finding the correct transaction that will yield the //
//  maximum profit, it makes greedy current (present-time)           //
//  optimizations at the expense of the overall optimization, and in //
// doing so, it avoids a time complexity containing n^2 because the  //
// current optimization avoids a nested loop that iterates through   //
// all 2d array indices.                                            //
//                                                                  //
//  ************************************************************  //

 //  Iterating through each stock to find the stock with the lowest value
 //  at index 0
 for (i = 0; i < predictedPrices.size(); i++)
 {

     if (predictedPrices[i][0] < smallestIndexZeroVALUE)
     {
         smallestIndexZeroVALUE = predictedPrices[i][0];
         smallestIndexZeroROW = i;
     }

 }
```

```
//  Iterating through the prices for each stock
for (j = 0; j < predictedPrices[smallestIndexZeroROW].size(); j++)
{

    //  I'm going through each previous price before the current
    //  stock price (price at index j) and subtracting each
    //  previous stock from the current stock. If the sum of
    //  that operation is larger than the current value of
    //  maxProfit, then the sum becomes the new value of
    //  maxProfit.
    for (k = 0; k < j; k++)
    {

        tempInt = predictedPrices[smallestIndexZeroROW][j] -
                  predictedPrices[smallestIndexZeroROW][k];

        //  If a new, larger value for maxProfit is found, we
        //  want to store and keep track of the new value for
        //  value for maxProfit as well as other information
        //  regarding that transaction for when we print the
        //  desired tuple of informational values.
        if (tempInt > maxProfit)
        {
            maxProfitStock = i;
            maxProfitBuyDay = k;
            maxProfitSellDay = j;
            maxProfit = tempInt;
        }

    }

}
```

```
    //  If we, indeed, found a max profit of some kind, then we want
    //  to add 1 to the values representing the stock row, buy day,
    //  and sell day. The reason is because the format given in the
    //  Final Project instructions gives examples where tuples
    //  contain values of the actual row and column number as opposed
    //  to programming index values (values that start with 0 i.e.
    //  0, 1, 2, 3, ...).
    if (maxProfit > 0)
    {
        maxProfitValues[0] = maxProfitStock + 1;
        maxProfitValues[1] = maxProfitBuyDay + 1;
        maxProfitValues[2] = maxProfitSellDay + 1;
        maxProfitValues[3] = maxProfit;
    }
    //  If maxProfit is 0, then no transaction was found that would
    //  lead to a profit. So we want to return the tuple (0, 0, 0, 0).
    else if (maxProfit <= 0)
    {
        maxProfitValues[0] = maxProfitStock;
        maxProfitValues[1] = maxProfitBuyDay;
        maxProfitValues[2] = maxProfitSellDay;
        maxProfitValues[3] = maxProfit;
    }

    return maxProfitValues;

}

//  Simply prints values to see if findMaxProfit() is working correctly
void printMaxProfitValues(vector<int> maxProfitValues, string vectorName)
{

    cout << endl;
    cout << "--------------------------------" << endl;
    cout << "\t" << vectorName << endl;
    cout << "--------------------------------" << endl;
```

```
    cout << endl;
    cout << "Tuple Output: " << endl;
    cout << endl;
    cout << "(" << maxProfitValues[0]   << ", "
                << maxProfitValues[1]   << ", "
                << maxProfitValues[2]   << ", "
                << maxProfitValues[3]   << ")";
    cout << endl;
    cout << endl;

}
```

Test Cases

```
////////////////////////////////////////////////////////////////////////
//                                                                    //
//                 STOCK       BUY DAY     SELL DAY    PROFIT       //
//      ANSWER:    1,          2,          5,          15           //
//                                                                    //
////////////////////////////////////////////////////////////////////////
vector<vector<int>> predictedPrices1
{

    {12, 1, 5, 3, 16},
    {4, 4, 13, 4, 9},
    {6, 8, 6, 1, 2},
    {14, 3, 4, 8, 10}

};

////////////////////////////////////////////////////////////////////////
//                                                                    //
//              STOCK       BUY DAY     SELL DAY    PROFIT           //
//  ANSWER:     4,          2,          3,          11               //
//                                                                    //
////////////////////////////////////////////////////////////////////////
vector<vector<int>> predictedPrices2
{

    {7, 1, 5, 3, 6},
    {2, 4, 3, 7, 9},
    {5, 8, 9, 1, 2},
    {9, 3, 14, 8, 7}

};

////////////////////////////////////////////////////////////////////////
//                                                                    //
//              STOCK       BUY DAY     SELL DAY    PROFIT           //
//  ANSWER:     0,          0,          0,          0                //
//                                                                    //
////////////////////////////////////////////////////////////////////////
vector<vector<int>> predictedPrices3
{

    {12, 8, 5, 3, 1},
    {20, 16, 14, 8, 7},
    {5, 4, 3, 2, 1},
    {0, 0, 0, 0, 0}

};
```

Test Results

```
PS C:\Users\jmmil\OneDrive\Desktop\Milestone3\Task2> mingw32-make
g++ -c -Wall -std=c++14 main.cpp
g++ main.o -o task2
PS C:\Users\jmmil\OneDrive\Desktop\Milestone3\Task2> mingw32-make run
./task2

---------------------------------
        predictedPrices1
---------------------------------

Tuple Output:

(5, 1, 3, 9)


---------------------------------
        predictedPrices2
---------------------------------

Tuple Output:

(5, 1, 5, 7)


---------------------------------
        predictedPrices3
---------------------------------

Tuple Output:

(0, 0, 0, 0)

PS C:\Users\jmmil\OneDrive\Desktop\Milestone3\Task2>
```

Description of the Implementation of the Algorithm:

A greedy algorithm will contain certain local optimizations at the expense of optimization for the entire problem to be solved as a whole. Taking that into consideration along with the time complexity of O(m * n) that implies a single loop followed by another novel, single loop, I designed an algorithm that found the lowest day 1 price for a stock and then iterated through that row in order to find the transaction within that stock row that yields the largest profit.

Limitations and Trade-offs:

Pros

- Although not as simplistic as a brute force approach, a greedy approach is also quite simple to implement.
- Comparatively time-efficient.
- These algorithms can often be a more time-efficient way to find a “close enough” solution to difficult problems.

Cons

- Local optimization can often be at the expense of global optimization. Therefore, local efficiency can lead to incorrect solutions.
- Greedy algorithms can often be not suitable for problems where the solution depends on the composition or size used as input.

Comparative Analysis:

When compared to task 1 utilizing a brute force approach, a greedy approach is far more time-efficient. When a greedy approach is compared to the dynamic programming of task 3, although a greedy approach will still be more efficient with time, a greedy approach does not guarantee an optimal solution. Also, when compared to dynamic programming, greedy algorithms are more efficient in terms of memory since there is no memoization.

Analysis of the Algorithm and Lessons Learned:

Greedy algorithms are quite useful for quick and possibly “close enough” solutions for problems where it’s appropriate. However, this can be situational as, often, “close enough" is not acceptable for accuracy problems that contain a specific emphasis on solution order.

Task - 3

Algorithm Implementation

```
#include <iostream>
#include <vector>
#include <string>

using namespace std;

vector<int> findMaxProfit(vector<vector<int>> predictedPrices);
void printMaxProfitValues(vector<int> maxProfitValues, string vectorName);

int recursiveStockMaxProfit(vector<int> daysVector, int currentDay, int
subtractedDay);

vector<int> maxProfitValues;

int main()
{

    //  I am declaring and initializing the 2D vectors to test the
    // implementation of the algorithm.

    ////////////////////////////////////////////////////////////////////
    //                                                                //
    //                 STOCK       BUY DAY     SELL DAY    PROFIT     //
    //      ANSWER:    1,          2,          5,          15         //
    //                                                                //
    ////////////////////////////////////////////////////////////////////
    vector<vector<int>> predictedPrices1
    {

        {12, 1, 5, 3, 16},
        {4, 4, 13, 4, 9},
        {6, 8, 6, 1, 2},
        {14, 3, 4, 8, 10}
    };
```

```
////////////////////////////////////////////////////////////////////////
//                                                                    //
//              STOCK       BUY DAY     SELL DAY    PROFIT            //
//  ANSWER:     4,          2,          3,          11                //
//                                                                    //
////////////////////////////////////////////////////////////////////////
vector<vector<int>> predictedPrices2
{

    {7, 1, 5, 3, 6},
    {2, 4, 3, 7, 9},
    {5, 8, 9, 1, 2},
    {9, 3, 14, 8, 7}

};

////////////////////////////////////////////////////////////////////////
//                                                                    //
//              STOCK       BUY DAY     SELL DAY    PROFIT            //
//  ANSWER:     0,          0,          0,          0                 //
//                                                                    //
////////////////////////////////////////////////////////////////////////
vector<vector<int>> predictedPrices3
{

    {12, 8, 5, 3, 1},
    {20, 16, 14, 8, 7},
    {5, 4, 3, 2, 1},
    {0, 0, 0, 0, 0}

};

maxProfitValues.resize(4, 0);
```

```
    //  I'm calling the findMaxProfit() function, which is the implementation
    //  of the algorithm, and the printMaxProfitValues() function that will
    //  confirm that the correct values are returned.
    vector<int> maxProfitValues1 = findMaxProfit(predictedPrices1);
    printMaxProfitValues(maxProfitValues1, "predictedPrices1");

    vector<int> maxProfitValues2 = findMaxProfit(predictedPrices2);
    printMaxProfitValues(maxProfitValues2, "predictedPrices2");

    vector<int> maxProfitValues3 = findMaxProfit(predictedPrices3);
    printMaxProfitValues(maxProfitValues3, "predictedPrices3");

    return 0;

}

//  Implementation of the algorithm.
vector<int> findMaxProfit(vector<vector<int>> predictedPrices)
{

    //  I'm declaring and initializing all necessary variables.
    long unsigned int i = 0;
    long unsigned int j = 0;

    int maxProfitStock = 0;
    int maxProfitBuyDay = 0;
    int maxProfitSellDay = 0;
    int maxProfit = 0;

    int buyPrice = 0;
    int buyPriceIndex = 0;

    //  I'm creating a vector of size 4 filled with 0 as all of its values.
    //  If a max profit from a transaction is, indeed, found, then the 0's
    //  will change to other values.
    vector<vector<int>> maxProfitStockDay(predictedPrices.size(),
      vector<int>(predictedPrices[0].size(), 0));
    vector<int> maxProfitValues(4, 0);
```

```
//  ************************************************************  //
//                                                               //
//  Because we're solving the same problem as the problem in Task 2  //
//  (including a lack of n^2 time complexity) while using a dynamic  //
//  algorithm instead, I would expect the resulting algorithm to     //
//  be similar to Task 2's algorithm but, however, contain           //
//  memoization which is the storing of results                      //
//  to subproblems so that they are immediately available without    //
//  the need for recalculation. This technique of memoization is     //
//  often implemented in cases where similar calculations are        //
//  repeatedly made like in the case of recursion. Because we are    //
//  repeatedly doing similar calculations when trying to find the    //
//  maximum price by subtracting previous stock prices from a        //
//  later (future) stock price, we want to implement memoization     //
//  here.                                                            //
//                                                               //
//  ************************************************************  //

//  Iterating through each stock
for (i = 0; i < predictedPrices.size(); i++)
{

    buyPrice = predictedPrices[i][0];
    buyPriceIndex = 0;

    //  Iterating through the prices for each stock
    for (j = 0; j < predictedPrices[i].size(); j++)
    {
        // Update minimum stock price encountered so far
        buyPrice = min(buyPrice, predictedPrices[i][j]);
        if (predictedPrices[i][j] < buyPrice)
        {
            buyPriceIndex = j;
        }
```

```
            //  If it's the first day for a given stock, then no profit is
            //  possible yet.
            if (j == 0)
            {
                maxProfitStockDay[i][j] = 0;
            }
            //  Otherwise, we want to see whether subtracting the lowest price
            //  found so far from the current stock price will yield a larger
            //  value that the current value found within the maxProfit
            //  variable. If it is larger, then we've found a new maxProfit.
            //  We do this until we ultimately find the largest profit
            // possible within the 2d vector.
            else
            {
                maxProfitStockDay[i][j] = max(maxProfitStockDay[i][j - 1],
                    predictedPrices[i][j] - buyPrice);

                if (predictedPrices[i][j] - buyPrice > maxProfit)
                {
                    maxProfitStock = i;
                    maxProfitBuyDay = buyPriceIndex + 1;
                    maxProfitSellDay = j;
                    maxProfit = predictedPrices[i][j] - buyPrice;

                }

            }
        }

    }
```

```
    //  If we, indeed, found a max profit of some kind, then we want
    //  to add 1 to the values representing the stock row, buy day,
    //  and sell day. The reason is because the format given in the
    //  Final Project instructions gives examples where tuples
    //  contain values of the actual row and column number as opposed
    //  to programming index values (values that start with 0 i.e.
    //  0, 1, 2, 3, ...).

    if (maxProfit > 0)
    {
        maxProfitValues[0] = maxProfitStock + 1;
        maxProfitValues[1] = maxProfitBuyDay + 1;
        maxProfitValues[2] = maxProfitSellDay + 1;
        maxProfitValues[3] = maxProfit;
    }
    //  If maxProfit is 0, then no transaction was found that would
    //  lead to a profit. So we want to return the tuple (0, 0, 0, 0).
    else if (maxProfit <= 0)
    {
        maxProfitValues[0] = maxProfitStock;
        maxProfitValues[1] = maxProfitBuyDay;
        maxProfitValues[2] = maxProfitSellDay;
        maxProfitValues[3] = maxProfit;
    }


    return maxProfitValues;

}
```

```
//  Simply prints values to see if findMaxProfit() is working correctly
void printMaxProfitValues(vector<int> maxProfitValues, string vectorName)
{

    cout << endl;
    cout << "---------------------------------" << endl;
    cout << "\t" << vectorName << endl;
    cout << "---------------------------------" << endl;

    cout << endl;
    cout << "Tuple Output: " << endl;
    cout << endl;
    cout << "(" << maxProfitValues[0]   << ", "
                << maxProfitValues[1]   << ", "
                << maxProfitValues[2]   << ", "
                << maxProfitValues[3]   << ")";
    cout << endl;
    cout << endl;

}
```

Test Cases

```
////////////////////////////////////////////////////////////////////////
//                                                                    //
//                 STOCK       BUY DAY     SELL DAY    PROFIT       //
//      ANSWER:    1,          2,          5,          15           //
//                                                                    //
////////////////////////////////////////////////////////////////////////
vector<vector<int>> predictedPrices1
{

    {12, 1, 5, 3, 16},
    {4, 4, 13, 4, 9},
    {6, 8, 6, 1, 2},
    {14, 3, 4, 8, 10}

};

////////////////////////////////////////////////////////////////////////
//                                                                    //
//             STOCK       BUY DAY     SELL DAY    PROFIT           //
//  ANSWER:    4,          2,          3,          11               //
//                                                                    //
////////////////////////////////////////////////////////////////////////
vector<vector<int>> predictedPrices2
{

    {7, 1, 5, 3, 6},
    {2, 4, 3, 7, 9},
    {5, 8, 9, 1, 2},
    {9, 3, 14, 8, 7}

};

////////////////////////////////////////////////////////////////////////
//                                                                    //
//             STOCK       BUY DAY     SELL DAY    PROFIT           //
//  ANSWER:    0,          0,          0,          0                //
//                                                                    //
////////////////////////////////////////////////////////////////////////
vector<vector<int>> predictedPrices3
{

    {12, 8, 5, 3, 1},
    {20, 16, 14, 8, 7},
    {5, 4, 3, 2, 1},
    {0, 0, 0, 0, 0}

};
```

Test Results

```
PS C:\Users\jmmil\OneDrive\Desktop\Milestone3\Task3> mingw32-make
g++ -c -Wall -std=c++14 main.cpp
g++ main.o -o task3
PS C:\Users\jmmil\OneDrive\Desktop\Milestone3\Task3> mingw32-make run
./task3

---------------------------------
        predictedPrices1
---------------------------------

Tuple Output:

(1, 2, 5, 15)


---------------------------------
        predictedPrices2
---------------------------------

Tuple Output:

(4, 2, 3, 11)


---------------------------------
        predictedPrices3
---------------------------------

Tuple Output:

(0, 0, 0, 0)

PS C:\Users\jmmil\OneDrive\Desktop\Milestone3\Task3>
```

Description of the Implementation of the Algorithm:

Because both a lack of n^2 found within the time complexity and the use of dynamic programming, I used the technique of memoization to keep a running maximum to store maxProfit in order to utilize previously computed results to solve the problem as a whole (use smaller subproblems to solve the greater solution).

Limitations and Trade-offs:

Pros

- Dynamic programming can be used for greater time efficiency and optimizations of problems that use regular recursion.
- Although a dynamic programming approach leads to greater time efficiency, it still yields an optimal, correct solution.

Cons

- Slower algorithm approach when compared to other algorithms.
- Can be more complicated to implement when compared to other algorithms.

Comparative Analysis:

When compared to brute force, the time efficiency of dynamic programming is greater. However, it is a more complicated form of algorithm to implement.

When compared to a greedy algorithm approach, it is guaranteed that dynamic programming will produce a correct, optimal solution compared to the possibly incorrect (“close enough”) solution from a greedy approach. Also, a dynamic approach uses previous calculations to ensure optimality while a greedy approach solves problems with local optimizations more in mind.

Analysis of the Algorithm and Lessons Learned:

Dynamic programming and the technique of memoization is an excellent approach to finding an accurate, correct solution like that of a brute force approach but with a noticeable increase in time efficiency.

Task - 6

Algorithm Implementation

```
#include <iostream>
#include <vector>
#include <string>

using namespace std;

vector<int> findMaxProfit(vector<vector<int>> predictedPrices, int c);
void printMaxProfitValues(vector<int> maxProfitValues, string vectorName);

int main()
{

    //  I am declaring and initializing the 2D vectors to test the
    //  implementation of the algorithm.

    ////////////////////////////////////////////////////////////
    //                                                        //
    //  MAX                                                   //
    //  PROFIT:     13                                        //
    //                                                        //
    ////////////////////////////////////////////////////////////
    vector<vector<int>> predictedPrices1
    {

        {7, 1, 5, 3, 6, 8, 9},
        {2, 4, 3, 7, 9, 1, 8},
        {5, 9, 9, 4, 2, 3, 7},
        {9, 3, 4, 8, 7, 4, 1},
        {3, 5, 5, 3, 1, 6, 10}

    };
    int c1 = 2;
```

```
//////////////////////////////////////////////////////////////
//                                                          //
//  MAX                                                     //
//  PROFIT:     16                                          //
//                                                          //
//////////////////////////////////////////////////////////////
vector<vector<int>> predictedPrices2
{

    {2, 9, 8, 4, 5, 0, 7},
    {6, 7, 3, 9, 1, 0, 8},
    {1, 7, 9, 6, 4, 9, 11},
    {7, 8, 3, 1, 8, 5, 2},
    {1, 8, 4, 0, 9, 2, 1}

};
int c2 = 2;

//////////////////////////////////////////////////////////////
//                                                          //
//  MAX                                                     //
//  PROFIT:     0                                           //
//                                                          //
//////////////////////////////////////////////////////////////
vector<vector<int>> predictedPrices3
{

    {12, 8, 5, 3, 1},
    {20, 16, 14, 8, 7},
    {5, 4, 3, 2, 1},
    {0, 0, 0, 0, 0}

};
int c3 = 2;
```

```
    //  I'm calling the findMaxProfit() function, which is the implementation
    //  of the algorithm, and the printMaxProfitValues() function that will
    //  confirm that the correct values are returned.
    vector<int> maxProfitValues1 = findMaxProfit(predictedPrices1, c1);
    printMaxProfitValues(maxProfitValues1, "predictedPrices1");

    vector<int> maxProfitValues2 = findMaxProfit(predictedPrices2, c2);
    printMaxProfitValues(maxProfitValues2, "predictedPrices2");

    vector<int> maxProfitValues3 = findMaxProfit(predictedPrices3, c3);
    printMaxProfitValues(maxProfitValues3, "predictedPrices3");

    return 0;

}

//  Implementation of the algorithm.
vector<int> findMaxProfit(vector<vector<int>> predictedPrices, int c)
{

    //  I'm declaring and initializing all necessary variables.
    long unsigned int i = 0;
    long unsigned int j = 0;
    long unsigned int k = 0;
    long unsigned int m = 0;

    int maxProfit = 0;
    int tempInt = 0;

    int lowestPrice = 0;
    int lowestPriceIndex = 0;
```

```
//  I'm creating a vector of size 4 filled with 0 as all of its values.
//  If a max profit from a transaction is, indeed, found, then the 0's
//  will change to other values.
vector<vector<int>> maxProfitEachStockDay(predictedPrices.size(),
  vector<int>(predictedPrices[0].size(), 0));

vector<vector<int>> buyDays(predictedPrices.size(),
  vector<int>(predictedPrices[0].size(), 0));

vector<int> maxProfitValues;

//  ************************************************************  //
//                                                               //
//  Because our time complexity contains n^2, we would expect that  //
//  our algorithm would contain a nested for loop. Therefore, we're  //
//  going to iterate through all elements of the 2d array containing  //
//  the stock prices by using a nested for loop, and because we're  //
//  utilizing dynamic programming, we're going to use recursion to  //
//  return the maximum profit selling a stock for each day by  //
//  subtracting previous stock prices from the current stock price,  //
//  comparing the current value of maxProfit with the profit value  //
//  returned from each recursion call, returning the larger of the  //
//  two values each time, and ultimately returning the max profit  //
//  possible for selling each stock on a particular day.  //
//                                                               //
//  ************************************************************  //

//  Iterating through each stock
for (i = 0; i < predictedPrices.size(); i++)
{

    maxProfit = 0;

    //  Iterating through the prices for each stock
    for (j = 0; j < predictedPrices[i].size(); j++)
    {
```

```
        lowestPrice = predictedPrices[i][0];
        lowestPriceIndex = 0;

        //  I'm going through each previous price before the current
        //  stock price (price at index j) and subtracting each
        //  previous stock from the current stock. If the sum of
        //  that operation is larger than the current value of
        //  maxProfit, then the sum becomes the new value of
        //  maxProfit.
        for (k = 0; k < j; k++)
        {

            //  I'm updating the lowest stock price found so far
            //  lowestPrice = min(lowestPrice, predictedPrices[i][j]);
            if (predictedPrices[i][k] < lowestPrice)
            {
                lowestPrice = predictedPrices[i][k];
                lowestPriceIndex = k;
            }

            tempInt = predictedPrices[i][j] - lowestPrice;

            //  If a new, larger value for maxProfit is found, we
            //  want to store and keep track of the new value for
            //  value for maxProfit as well as other information
            //  regarding that transaction for when we print the
            //  desired tuple of informational values.
            if (tempInt > maxProfit)
            {
                maxProfit = tempInt;
            }

        }
```

```
            maxProfitEachStockDay[i][j] = maxProfit;
            buyDays[i][j] = lowestPriceIndex;

        }

    }

    int maxProfitOverall = 0;

    int stock1 = 0;
    int buyDay1 = 0;
    int sellDay1 = 0;

    int stock2 = 0;
    int buyDay2 = 0;
    int sellDay2 = 0;

    //  Iterating through each stock
    for (i = 0; i < maxProfitEachStockDay.size(); i++)
    {

        //  Iterating through each day for each stock
        for (j = 0; j < maxProfitEachStockDay[i].size(); j++)
        {

            //  Now I'm going to only iterate through each stock that can
            //  validly make a second transaction when considering the
            //  constraint c
            for (m = 0; m < maxProfitEachStockDay.size(); m++)
            {
```

```
            //  Now I'm going to only iterate through each day for each
            //  stock that can validly make a second transaction when
            //  considering the constraint c
            for (k = j + 1 + 1 + c + 1; k <
                 maxProfitEachStockDay[i].size(); k++)
            {

                //  If the sum of the two transaction can produce a
                //  larger value than the current value of
                //  maxProfitOverall, then we replace maxProfitValue
                //  with the new sum.
                if ( ((maxProfitEachStockDay[i][j] +
                       maxProfitEachStockDay[m][k]) > maxProfitOverall)
                        && ( (buyDays[m][k] - (int)j) >= (c - 2) ) )
                {

                    maxProfitOverall = maxProfitEachStockDay[i][j] +
                    maxProfitEachStockDay[m][k];

                    stock1 = i;
                    buyDay1 = buyDays[i][j];
                    sellDay1 = j;

                    stock2 = m;
                    buyDay2 = buyDays[m][k];
                    sellDay2 = k;

                }

            }

        }

    }

}
```

```
    //  If we, indeed, found a max profit of some kind, then we want
    //  to add 1 to the values representing the stock row, buy day,
    //  and sell day. The reason is because the format given in the
    //  Final Project instructions gives examples where tuples
    //  contain values of the actual row and column number as opposed
    //  to programming index values (values that start with 0 i.e.
    //  0, 1, 2, 3, ...).
    if (maxProfitOverall > 0)
    {

        maxProfitValues.push_back(maxProfitOverall);

        maxProfitValues.push_back(stock1 + 1);
        maxProfitValues.push_back(buyDay1 + 1);
        maxProfitValues.push_back(sellDay1 + 1);

        maxProfitValues.push_back(stock2 + 1);
        maxProfitValues.push_back(buyDay2 + 1);
        maxProfitValues.push_back(sellDay2 + 1);

    }
    //  If maxProfit is 0, then no transaction was found that would
    //  lead to a profit. So we want to return the tuple (0, 0, 0, 0).
    else if (maxProfitOverall <= 0)
    {

        maxProfitValues.push_back(0);
        maxProfitValues.push_back(0);
        maxProfitValues.push_back(0);
        maxProfitValues.push_back(0);

    }

    return maxProfitValues;

}
```

```
//  Simply prints values to see if findMaxProfit() is working correctly
void printMaxProfitValues(vector<int> maxProfitValues, string vectorName)
{

    cout << endl;
    cout << "---------------------------------" << endl;
    cout << "\t" << vectorName << endl;
    cout << "---------------------------------" << endl;

    cout << endl;
    cout << "Tuple Output: " << endl;
    cout << endl;

    for (long unsigned int i = 0; i < maxProfitValues.size(); i++)
    {

        if (i == 0)
        {
            cout << "maxProfitOverall: " << maxProfitValues[i] << endl;
        }
        else if ( ((i - 1) % 3) == 0)
        {
            cout << "(" << maxProfitValues[i]   << ", ";
        }
        else if ( (i % 3) == 0 && i != 1 && (i + 1) != maxProfitValues.size())
        {
            cout << maxProfitValues[i] << "), " << endl;
        }
        else if ( (i % 3) == 0 && i != 1)
        {
            cout << maxProfitValues[i] << ")" << endl;
        }
        else
        {
            cout << maxProfitValues[i]   << ", ";
        }
    }
```

```
    cout << endl;
    cout << endl;

}
```

Test Cases

```
//////////////////////////////////////////////////////////////
//                                                          //
//  MAX                                                     //
//  PROFIT:     13                                          //
//                                                          //
//////////////////////////////////////////////////////////////
vector<vector<int>> predictedPrices1
{

    {7, 1, 5, 3, 6, 8, 9},
    {2, 4, 3, 7, 9, 1, 8},
    {5, 9, 9, 4, 2, 3, 7},
    {9, 3, 4, 8, 7, 4, 1},
    {3, 5, 5, 3, 1, 6, 10}

};
int c1 = 2;

//////////////////////////////////////////////////////////////
//                                                          //
//  MAX                                                     //
//  PROFIT:     16                                          //
//                                                          //
//////////////////////////////////////////////////////////////
vector<vector<int>> predictedPrices2
{

    {2, 9, 8, 4, 5, 0, 7},
    {6, 7, 3, 9, 1, 0, 8},
    {1, 7, 9, 6, 4, 9, 11},
    {7, 8, 3, 1, 8, 5, 2},
    {1, 8, 4, 0, 9, 2, 1}

};
int c2 = 2;

//////////////////////////////////////////////////////////////
//                                                          //
//  MAX                                                     //
//  PROFIT:     0                                           //
//                                                          //
//////////////////////////////////////////////////////////////
vector<vector<int>> predictedPrices3
{

    {12, 8, 5, 3, 1},
    {20, 16, 14, 8, 7},
    {5, 4, 3, 2, 1},
    {0, 0, 0, 0, 0}

};
int c3 = 2;
```

Test Results

```
PS C:\Users\jmmil\OneDrive\Desktop\Milestone3\Task6> mingw32-make
g++ -c -Wall -std=c++14 main.cpp
g++ main.o -o task6
PS C:\Users\jmmil\OneDrive\Desktop\Milestone3\Task6> mingw32-make run
./task6

--------------------------------
        predictedPrices1
--------------------------------

Tuple Output:

maxProfitOverall: 13
(3, 1, 2),
(5, 5, 7)



--------------------------------
        predictedPrices2
--------------------------------

Tuple Output:

maxProfitOverall: 16
(1, 1, 2),
(5, 4, 7)



--------------------------------
        predictedPrices3
--------------------------------

Tuple Output:

maxProfitOverall: 0
(0, 0, 0)


PS C:\Users\jmmil\OneDrive\Desktop\Milestone3\Task6>
```

Description of the Implementation of the Algorithm:

The given time complexity contains n^2, so an implementation of the algorithm to solve this problem could contain a nested for loop. Therefore, I iterated through all elements of the 2d array containing the stock prices by using a nested for loop, and because we're utilizing dynamic programming, recursion is used to return the maximum profit selling a stock for each day by subtracting previous stock prices from the current stock price, comparing the current value of maxProfit with the profit value returned from each recursion call, returning the larger of the two values each time, and ultimately returning the max profit possible for selling each stock on a particular day.

Limitations and Trade-offs:

Pros

- Dynamic programming can be used for greater time efficiency and optimizations of problems that use regular recursion.
- Although a dynamic programming approach leads to greater time efficiency, it still yields an optimal, correct solution.

Cons

- Slower algorithm approach when compared to other algorithms.
- Can be more complicated to implement when compared to other algorithms.

Comparative Analysis:

When compared to brute force, the time efficiency of dynamic programming is greater. However, it is a more complicated form of algorithm to implement.

When compared to a greedy algorithm approach, it is guaranteed that dynamic programming will produce a correct, optimal solution compared to the possibly incorrect ("close enough") solution from a greedy approach. Also, a dynamic approach uses previous calculations to ensure optimality while a greedy approach solves problems with local optimizations more in mind.

Analysis of the Algorithm and Lessons Learned:

Dynamic programming and the technique of memoization is an excellent approach to finding an accurate, correct solution like that of a brute force approach but with a noticeable increase in time efficiency.

Task - 7

Algorithm Implementation

```cpp
#include <iostream>
#include <vector>
#include <string>

using namespace std;

vector<int> findMaxProfit(vector<vector<int>> predictedPrices, int c);
void printMaxProfitValues(vector<int> maxProfitValues, string vectorName);

int main()
{

    //  I am declaring and initializing the 2D vectors to test the
    //  implementation of the algorithm.

    ////////////////////////////////////////////////////////////
    //                                                        //
    //  MAX                                                   //
    //  PROFIT:     13                                        //
    //                                                        //
    ////////////////////////////////////////////////////////////
    vector<vector<int>> predictedPrices1
    {

        {7, 1, 5, 3, 6, 8, 9},
        {2, 4, 3, 7, 9, 1, 8},
        {5, 9, 9, 4, 2, 3, 7},
        {9, 3, 4, 8, 7, 4, 1},
        {3, 5, 5, 3, 1, 6, 10}

    };
    int c1 = 2;
```

```
////////////////////////////////////////////////////////////////
//                                                            //
//  MAX                                                       //
//  PROFIT:     16                                            //
//                                                            //
////////////////////////////////////////////////////////////////
vector<vector<int>> predictedPrices2
{

    {2, 9, 8, 4, 5, 0, 7},
    {6, 7, 3, 9, 1, 0, 8},
    {1, 7, 9, 6, 4, 9, 11},
    {7, 8, 3, 1, 8, 5, 2},
    {1, 8, 4, 0, 9, 2, 1}

};
int c2 = 2;

////////////////////////////////////////////////////////////////
//                                                            //
//  MAX                                                       //
//  PROFIT:     0                                             //
//                                                            //
////////////////////////////////////////////////////////////////
vector<vector<int>> predictedPrices3
{

    {12, 8, 5, 3, 1},
    {20, 16, 14, 8, 7},
    {5, 4, 3, 2, 1},
    {0, 0, 0, 0, 0}

};
int c3 = 2;
```

```
    //  I'm calling the findMaxProfit() function, which is the implementation
    //  of the algorithm, and the printMaxProfitValues() function that will
    //  confirm that the correct values are returned.
    vector<int> maxProfitValues1 = findMaxProfit(predictedPrices1, c1);
    printMaxProfitValues(maxProfitValues1, "predictedPrices1");

    vector<int> maxProfitValues2 = findMaxProfit(predictedPrices2, c2);
    printMaxProfitValues(maxProfitValues2, "predictedPrices2");

    vector<int> maxProfitValues3 = findMaxProfit(predictedPrices3, c3);
    printMaxProfitValues(maxProfitValues3, "predictedPrices3");

    return 0;
}

//  Implementation of the algorithm.
vector<int> findMaxProfit(vector<vector<int>> predictedPrices, int c)
{

    //  I'm declaring and initializing all necessary variables.
    long unsigned int i = 0;
    long unsigned int j = 0;
    long unsigned int k = 0;
    long unsigned int m = 0;

    int maxProfit = 0;
    int tempInt = 0;

    int lowestPrice = 0;
    int lowestPriceIndex = 0;

    //  I'm creating a vector of size 4 filled with 0 as all of its values.
    //  If a max profit from a transaction is, indeed, found, then the 0's
    //  will change to other values.
    vector<vector<int>> maxProfitEachStockDay(predictedPrices.size(),
      vector<int>(predictedPrices[0].size(), 0));
```

```
vector<vector<int>> buyDays(predictedPrices.size(),
  vector<int>(predictedPrices[0].size(), 0));

vector<int> maxProfitValues;

//  ************************************************************  //
//                                                               //
//  This algorithm will be almost exactly the same as the algorithm  //
//  created in Task 6. However, with a time complexity of O(m · n),  //
//  we are going to replace the nested for loop with recursive calls  //
//  within each row for each possible pair of a future and previous  //
//  stock price that could yield a profit.                      //
//                                                               //
//  ************************************************************  //


//  Iterating through each stock
for (i = 0; i < predictedPrices.size(); i++)
{

    maxProfit = 0;

    //  Iterating through the prices for each stock
    for (j = 0; j < predictedPrices[i].size(); j++)
    {

        lowestPrice = predictedPrices[i][0];
        lowestPriceIndex = 0;
```

```
            //  I'm going through each previous price before the current
            //  stock price (price at index j) and subtracting each
            //  previous stock from the current stock. If the sum of
            //  that operation is larger than the current value of
            //  maxProfit, then the sum becomes the new value of
            //  maxProfit.
            for (k = 0; k < j; k++)
            {

                //  I'm updating the lowest stock price found so far
                if (predictedPrices[i][k] < lowestPrice)
                {
                    lowestPrice = predictedPrices[i][k];
                    lowestPriceIndex = k;
                }

                tempInt = predictedPrices[i][j] - lowestPrice;

                //  If a new, larger value for maxProfit is found, we
                //  want to store and keep track of the new value for
                //  value for maxProfit as well as other information
                //  regarding that transaction for when we print the
                //  desired tuple of informational values.
                if (tempInt > maxProfit)
                {
                    maxProfit = tempInt;
                }

            }

            maxProfitEachStockDay[i][j] = maxProfit;
            buyDays[i][j] = lowestPriceIndex;

        }

    }
```

```
int maxProfitOverall = 0;

int stock1 = 0;
int buyDay1 = 0;
int sellDay1 = 0;

int stock2 = 0;
int buyDay2 = 0;
int sellDay2 = 0;

//  Iterating through each stock
for (i = 0; i < maxProfitEachStockDay.size(); i++)
{

    //  Iterating through each day for each stock
    for (j = 0; j < maxProfitEachStockDay[i].size(); j++)
    {

        //  Now I'm going to only iterate through each stock that can
        //  validly make a second transaction when considering the
        //  constraint c
        for (m = 0; m < maxProfitEachStockDay.size(); m++)
        {

            //  Now I'm going to only iterate through each day for each
            //  stock that can validly make a second transaction when
            //  considering the constraint c
            for (k = j + 1 + 1 + c + 1; k <
                 maxProfitEachStockDay[i].size(); k++)
            {
```

```
                //  If the sum of the two transaction can produce a
                //  larger value than the current value of
                //  maxProfitOverall, then we replace maxProfitValue
                //  with the new sum.
                if ( ((maxProfitEachStockDay[i][j] +
                       maxProfitEachStockDay[m][k]) > maxProfitOverall)
                        && ( (buyDays[m][k] - (int)j) >= (c - 2) ) )
                {

                    maxProfitOverall = maxProfitEachStockDay[i][j] +
                    maxProfitEachStockDay[m][k];

                    stock1 = i;
                    buyDay1 = buyDays[i][j];
                    sellDay1 = j;

                    stock2 = m;
                    buyDay2 = buyDays[m][k];
                    sellDay2 = k;

                }

            }

        }

    }

}
```

```
    //  If we, indeed, found a max profit of some kind, then we want
    //  to add 1 to the values representing the stock row, buy day,
    //  and sell day. The reason is because the format given in the
    //  Final Project instructions gives examples where tuples
    //  contain values of the actual row and column number as opposed
    //  to programming index values (values that start with 0 i.e.
    //  0, 1, 2, 3, ...).
    if (maxProfitOverall > 0)
    {

        maxProfitValues.push_back(maxProfitOverall);

        maxProfitValues.push_back(stock1 + 1);
        maxProfitValues.push_back(buyDay1 + 1);
        maxProfitValues.push_back(sellDay1 + 1);

        maxProfitValues.push_back(stock2 + 1);
        maxProfitValues.push_back(buyDay2 + 1);
        maxProfitValues.push_back(sellDay2 + 1);

    }
    //  If maxProfit is 0, then no transaction was found that would
    //  lead to a profit. So we want to return the tuple (0, 0, 0, 0).
    else if (maxProfitOverall <= 0)
    {

        maxProfitValues.push_back(0);
        maxProfitValues.push_back(0);
        maxProfitValues.push_back(0);
        maxProfitValues.push_back(0);

    }

    return maxProfitValues;

}
```

```
//  Simply prints values to see if findMaxProfit() is working correctly
void printMaxProfitValues(vector<int> maxProfitValues, string vectorName)
{

    cout << endl;
    cout << "---------------------------------" << endl;
    cout << "\t" << vectorName << endl;
    cout << "---------------------------------" << endl;

    cout << endl;
    cout << "Tuple Output: " << endl;
    cout << endl;

    for (long unsigned int i = 0; i < maxProfitValues.size(); i++)
    {

        if (i == 0)
        {
            cout << "maxProfitOverall: " << maxProfitValues[i] << endl;
        }
        else if ( ((i - 1) % 3) == 0)
        {
            cout << "(" << maxProfitValues[i]   << ", ";
        }
        else if ( (i % 3) == 0 && i != 1 && (i + 1) != maxProfitValues.size())
        {
            cout << maxProfitValues[i] << "), " << endl;
        }
        else if ( (i % 3) == 0 && i != 1)
        {
            cout << maxProfitValues[i] << ")" << endl;
        }
        else
        {
            cout << maxProfitValues[i]   << ", ";
        }
    }
```

```
    cout << endl;
    cout << endl;

}
```

Test Cases

```
////////////////////////////////////////////////////////////////
//                                                            //
//  MAX                                                       //
//  PROFIT:     13                                            //
//                                                            //
////////////////////////////////////////////////////////////////
vector<vector<int>> predictedPrices1
{

    {7, 1, 5, 3, 6, 8, 9},
    {2, 4, 3, 7, 9, 1, 8},
    {5, 9, 9, 4, 2, 3, 7},
    {9, 3, 4, 8, 7, 4, 1},
    {3, 5, 5, 3, 1, 6, 10}

};
int c1 = 2;

////////////////////////////////////////////////////////////////
//                                                            //
//  MAX                                                       //
//  PROFIT:     16                                            //
//                                                            //
////////////////////////////////////////////////////////////////
vector<vector<int>> predictedPrices2
{

    {2, 9, 8, 4, 5, 0, 7},
    {6, 7, 3, 9, 1, 0, 8},
    {1, 7, 9, 6, 4, 9, 11},
    {7, 8, 3, 1, 8, 5, 2},
    {1, 8, 4, 0, 9, 2, 1}

};
int c2 = 2;

////////////////////////////////////////////////////////////////
//                                                            //
//  MAX                                                       //
//  PROFIT:     0                                             //
//                                                            //
////////////////////////////////////////////////////////////////
vector<vector<int>> predictedPrices3
{

    {12, 8, 5, 3, 1},
    {20, 16, 14, 8, 7},
    {5, 4, 3, 2, 1},
    {0, 0, 0, 0, 0}

};
int c3 = 2;
```

Test Results

```
PS C:\Users\jmmil\OneDrive\Desktop\Milestone3\Task7> mingw32-make
g++ -c -Wall -std=c++14 main.cpp
g++ main.o -o task7
PS C:\Users\jmmil\OneDrive\Desktop\Milestone3\Task7> mingw32-make run
./task7

--------------------------------
        predictedPrices1
--------------------------------

Tuple Output:

maxProfitOverall: 13
(3, 1, 2),
(5, 5, 7)



--------------------------------
        predictedPrices2
--------------------------------

Tuple Output:

maxProfitOverall: 16
(1, 1, 2),
(5, 4, 7)



--------------------------------
        predictedPrices3
--------------------------------

Tuple Output:

maxProfitOverall: 0
(0, 0, 0)


PS C:\Users\jmmil\OneDrive\Desktop\Milestone3\Task7>
```

Description of the Implementation of the Algorithm:

Similar to the algorithm created in Task 6. However, with a time complexity of $O(m \cdot n)$, we are going to replace the nested for loop with recursive calls within each row for each possible pair of a future and previous stock price that could yield a profit.

Limitations and Trade-offs:

Pros

- Dynamic programming can be used for greater time efficiency and optimizations of problems that use regular recursion.
- Although a dynamic programming approach leads to greater time efficiency, it still yields an optimal, correct solution.

Cons

- Slower algorithm approach when compared to other algorithms.
- Can be more complicated to implement when compared to other algorithms.

Comparative Analysis:

When compared to brute force, the time efficiency of dynamic programming is greater. However, it is a more complicated form of algorithm to implement.

When compared to a greedy algorithm approach, it is guaranteed that dynamic programming will produce a correct, optimal solution compared to the possibly incorrect (“close enough”) solution from a greedy approach. Also, a dynamic approach uses previous calculations to ensure optimality while a greedy approach solves problems with local optimizations more in mind.

Analysis of the Algorithm and Lessons Learned:

Dynamic programming and the technique of memoization is an excellent approach to finding an accurate, correct solution like that of a brute force approach but with a noticeable increase in time efficiency.