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

Milestone 1

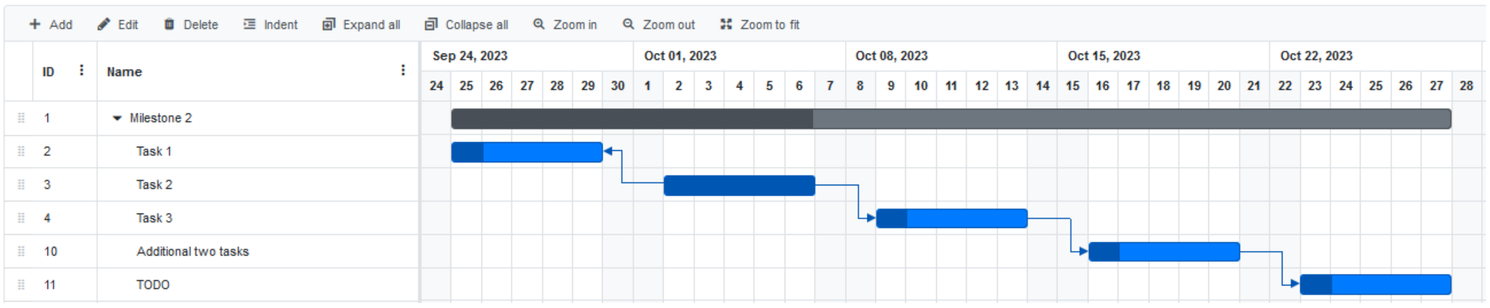
Github Repository Link: https://github.com/jackmillsgator/COP4533FinalProject

Group Members: Jack Mills UFID3485-7973

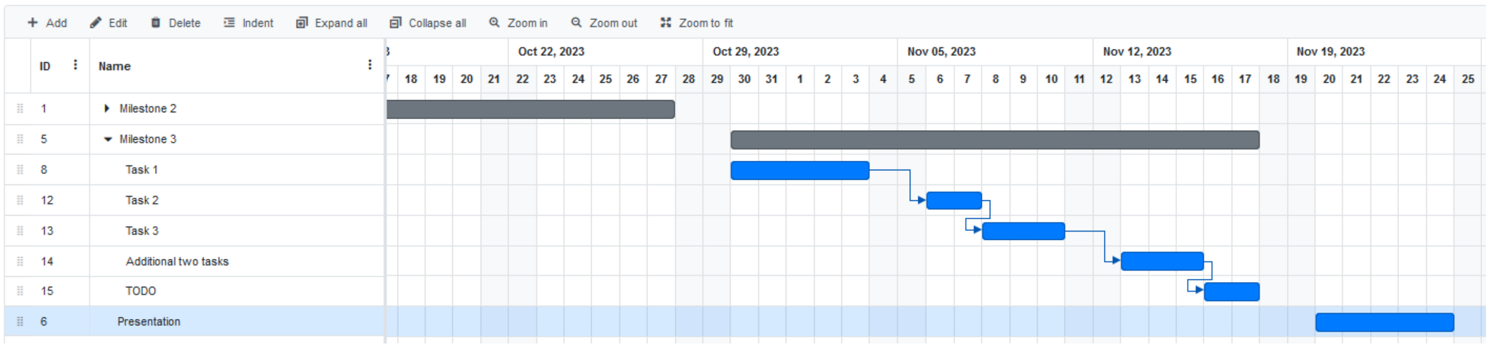
Member Roles: I, Jack Mills, am the sole member of my group. So, I'm assuming all Final Project roles.

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

Project Gantt Chart:



ID	Name
1	Milestone 2
2	Task 1
3	Task 2
4	Task 3
10	Additional two tasks
11	TODO



ID	Name
1	Milestone 2
5	Milestone 3
8	Task 1
12	Task 2
13	Task 3
14	Additional two tasks
15	TODO
6	Presentation

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

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 · 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 · 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 · 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.
```