COP4533 Algorithms, Abstraction, and Design
Project Milestone 3

1 Group Members

Jonathan Williams

2 Member Roles

Jonathan Williams - Work through problem sets for all milestones. Transcribe solutions for submission on a pdf, and maintain the project github page with any updates.

3 Communication Methods

Since there is only one member there will not be a main form of communication. Progress will be tracked using the project Gantt chart and the Canvas calendar.

4 Project Gantt Chart

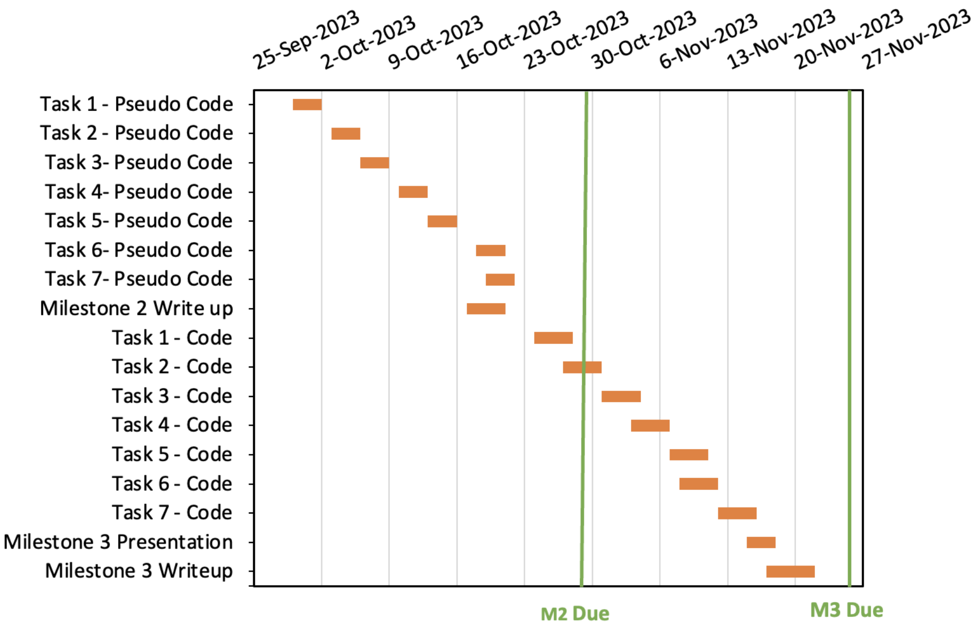

Figure 1: Proposed progress of future milestones.

5 GitHub Repository Link

https://github.com/Jon-Williams/COP4533Project

6 Programming Language

All code solutions will be done using Python.

7 Problem 1

7.1 Manual Solution

We begin with the following matrix, A, representing four stock prices over a five day period.

$$A = \begin{bmatrix} 12 & 1 & 5 & 3 & 16 \\ 4 & 4 & 13 & 4 & 9 \\ 6 & 8 & 6 & 1 & 2 \\ 14 & 3 & 4 & 8 & 10 \end{bmatrix}$$

First we calculated all possible single transactions, to see which single transaction had the maximum profit per stock.

Output \ Stock Index	1	2	3	4
1	(1, 1, 2, -11)	(2, 1, 2, 0)	(3, 1, 2, 2)	(4, 1, 2, -11)
2	(1, 1, 3, -7)	(2, 1, 3, 9)	(3, 1, 3, 0)	(4, 1, 3, -10)
3	(1, 1, 4, -9)	(2, 1, 4, 0)	(3, 1, 4, -5)	(4, 1, 4, -6)
4	(1, 1, 5, 4)	(2, 1, 5, 5)	(3, 1, 5, -4)	(4, 1, 5, -4)
5	(1, 2, 3, 4)	(2, 2, 3, 9)	(3, 2, 3, -2)	(4, 2, 3, 1)
6	(1, 2, 4, 2)	(2, 2, 4, 0)	(3, 2, 4, -7)	(4, 2, 4, 5)
7	(1, 2, 5, 15)	(2, 2, 5, 5)	(3, 2, 5, -6)	(4, 2, 5, 7)
8	(1, 3, 4, -2)	(2, 3, 4, -9)	(3, 3, 4, -5)	(4, 3, 4, 4)
9	(1, 3, 5, 11)	(2, 3, 5, -4)	(3, 3, 5, -4)	(4, 3, 5, 6)
10	(1, 4, 5, 13)	(2, 4, 5, 5)	(3, 4, 5, 1)	(4, 4, 5, 2)
Max Transaction Profit	(1, 2, 5, 15)	(2, 1, 3, 9)	(3, 1, 2, 2)	(4, 2, 5, 7)

The single transaction that will lead the maximum profit is (1, 2, 5, 15), buying stock 1 on day 2 and selling on day 5 with a profit of 15.

7.2 Task 1. Brute Force Solution To Problem 1

The following is the proposed brute force algorithm to solve problem 1. This implementation will iterate over each of the stocks once, it will then iterate over each day except the last one as a 'buy day', and all remaining days as a 'sell day'. Therefore we have a time complexity of

$$O(stock * days^2)$$

7.2.1 Assumptions and Definitions

First we assume everything is 1 indexed, since the solution will be 1 indexed. Next, we assume that a function size() exists which will return the dimensions of a given matrix input and return a two element array of the row and column size of the matrix.

7.2.2 Pseudo code

procedure SingleTransactionBruteForce(Stock Matrix = A)
    maxProfit ← -0
    solutionArray ← [0, 0, 0, 0]
    [rowDim, colDim] ← size(A)
    for stockIndex = 1 to rowDim **do**
        for buyDayIndex = 1 to colDim - 1 **do**
            for sellDayIndex = buyDayIndex + 1 to colDim **do**
                buyPrice ← A[stockIndex][buyDayIndex]
                sellPrice ← A[stockIndex][sellDayIndex]
                profit ← sellPrice - buyPrice
                if profit > maxProfit **then**
                    maxProfit ← profit
                    solutionArray ← [stockIndex, buyDayIndex, sellDayIndex, profit]
                end if
            end for
        end for
    end for
    return solutionArray
end procedure

7.2.3 Python Implementation

```python
import numpy as np

def single_transaction_brute_force(stock_matrix):
    max_profit = -float('inf')
    solution_array = [0, 0, 0, 0]
    row_dim, col_dim = np.shape(stock_matrix)

    for stock_index in range(row_dim):
        for buy_day_index in range(col_dim - 1):
            for sell_day_index in range(buy_day_index + 1, col_dim):
                buy_price = stock_matrix[stock_index][buy_day_index]
                sell_price = stock_matrix[stock_index][sell_day_index]
                profit = sell_price - buy_price
```

```
                if profit > max_profit:
                    max_profit = profit
                    solution_array = [stock_index, buy_day_index, sell_day_
    
    return solution_array
```

7.2.4 Description, Analysis, and Limitations, for Task 1

This brute force implementation iterates through all possible combinations of days and calculates the profit for all these buy/sell combinations. The biggest limitation of the brute force algorithm is it's inefficiency at the cost of a simple implementation. Although it is simple to implement the solution the time complexity of O(m*n$\hat{2}$) will be unacceptable as number of days being considered increases.

7.3 Task 2. Greedy Solution To Problem 1

The following is the proposed greedy approach for Problem 1. This approach will first iterate over every stock, and then iterate over each day. Unlike the brute force implementation, this approach will only iterate over the days once. We can get away with iterating over days once by trying to make a greedy choice throughout. The greedy choice will be made by logging the minimum price of the stock seen so far, next it will calculate the potential profit of selling in each day given we bought at the minimum price seen so far of the stock. Since we only iterate over the days once per stock we have a time complexity of

$$O(stock * days)$$

7.3.1 Assumption and Definitions

Like before we assume that the stock matrix is 1 indexed, and there exists a size() function that can take in the 2D matrix A and return an array with the row and column dimensions. Finally we can define '-Inf' to be the smallest number possible such that for any number x, such that:

$$\forall x \in R : x > -Inf = True$$

7.3.2 Pseudo Code

procedure SingleTransactionGreedy(Stock Matrix = A)
    maxProfit ← 0
    solutionArray ← [0, 0, 0, 0]
    [rowDim, colDim] ← size(A)
    for stockIndex = 1 to rowDim **do**
        minPrice ← Inf
        minPriceDate ← 0
        potentialProfit ← 0
        for dayIndex = 1 to colDim **do**

currentPrice ← A[stockIndex][dayIndex]
if minPrice > currentPrice **then**
minPrice ← currentPrice)
minPriceDate ← dayIndex
end if
potentialProfit ← currentPrice - minPrice
if maxProfit < potentialProfit **then**
maxProfit ← potentialProfit
solutionArray ← [stockIndex, minPriceDate, dayIndex, maxProfit]
end if
end for
end for
return solutionArray
end procedure

7.3.3 Python Implementation

```
import numpy as np

def single_transaction_greedy(stock_matrix):
    max_profit = 0
    solution_array = [0, 0, 0, 0]
    row_dim, col_dim = np.shape(stock_matrix)

    for stock_index in range(row_dim):
        min_price = float('inf')
        min_price_date = 0
        potential_profit = 0

        for day_index in range(col_dim):
            current_price = stock_matrix[stock_index][day_index]

            if current_price < min_price:
                min_price = current_price
                min_price_date = day_index

            potential_profit = current_price - min_price

            if potential_profit > max_profit:
                max_profit = potential_profit
                solution_array = [stock_index, min_price_date, ...
                day_index, max_profit]

    return solution_array
```

7.3.4 Description, Analysis, and Limitations, for Task 2

This greedy approach to the simple case of one transaction, iterates through each of the stocks while keeping track of the minimum price of a stock seen. The algorithm is greedy as it will log the cheapest price seen so far, and will check if the potential profit is higher each iteration. With a time complexity of O(m*n), it provides a huge performance benefit over the brute force approach. A greedy approach, although fast, does not guarantee to get the maximum profit, and could land on a sub optimal solution. This choice is the best when the global optimal solution is not needed and fast computation is desired.

7.4 Task 3. Dynamic Programming Solution To Problem 1

In this implementation we utilize dynamic programming as a sliding window approach to make the global best decision per stock. Like the previous example we only iterate over the number of days once per stock. During the iterations of the predicted prices, we keep track of the lowest price possible for each purchase and we check the potential profit for selling at every particular day. This guarantees an optimal solution with a run time complexity of

$$O(stock * days)$$

7.4.1 Assumption and Definitions

As always we assume an indexing of 1 for all arrays and matrices, and a function size() which will output the resulting dimensions as integers of a given input matrix. In this implementations we also assume that the given matrix A will have at least two columns such that there is one day to buy and one to sell.

7.4.2 Pseudo Code

```
procedure SingleTransactionDynamic(Stock Matrix = A)
    maxProfit ← 0
    solutionArray ← [0, 0, 0, 0]
    [rowDim, colDim] ← size(A)
    for stockIndex = 1 to rowDim do
        potentialProfit ← 0
        buyDay ← 1
        for sellDay = 2 to colDim do
            buyPrice ← A[stockIndex][buyDay]
            sellPrice ← A[stockIndex][sellDay]
            if buyPrice < sellPrice then
                potentialProfit ← sellPrice - buyPrice
                if maxProfit < potentialProfit then
                    maxProfit ← potentialProfit
                    solutionArray ← [stockIndex, buyDay, sellDay, maxProfit]
                end if
```

```
            else
                buyDay ← sellDay
            end if
        end for
    end for
    return solutionArray
end procedure
```

7.4.3 Python Implementation

```
import numpy as np

def single_transaction_dynamic(stock_matrix):
    max_profit = 0
    solution_array = [0, 0, 0, 0]
    row_dim, col_dim = np.shape(stock_matrix)

    for stock_index in range(row_dim):
        potential_profit = 0
        buy_day = 0

        for sell_day in range(1, col_dim):
            buy_price = stock_matrix[stock_index][buy_day]
            sell_price = stock_matrix[stock_index][sell_day]

            if buy_price < sell_price:
                potential_profit = sell_price - buy_price

                if potential_profit > max_profit:
                    max_profit = potential_profit
                    solution_array = [stock_index, buy_day, ...
                    sell_day, max_profit]
            else:
                buy_day = sell_day

    return solution_array
```

7.4.4 Description, Analysis, and Limitations, for Task 3

For the final implementation of the single transaction case, Problem 1, we used a dynamic programming approach. We utilized a list structure to crate a dynamic table, which stores the maximum amount of profit which can be obtained for each transaction. Looking at previous prices on the table it can calculate the price at the current iteration, giving a time complexity of O(m*n). One drawback of this approach is that it is much more memory intensive compared to the greedy approach, since all possible profits are stored on the dynamic table.

8 Problem 2

8.1 Manual Solution

Next, we are given matrix A, with up to K transactions.

$$A = \begin{bmatrix} 25 & 30 & 15 & 40 & 50 \\ 10 & 20 & 30 & 25 & 5 \\ 30 & 45 & 35 & 10 & 15 \\ 5 & 50 & 35 & 25 & 45 \end{bmatrix}$$

The same as before, we calculated the potential profit for each of the stocks. Next, transactions that resulted in no profit were omitted, and finally the results were sorted in ascending profit.

Output \ Stock Index	1	2	3	4
1	(1, 1, 2, 5)	(2, 2, 4, 5)	(3, 1, 3, 5)	(4, 3, 5, 10)
2	(1, 2, 4, 10)	(2, 1, 2, 10)	(3, 4, 5, 5)	(4, 1, 4, 20)
3	(1, 4, 5, 10)	(2, 2, 3, 10)	(3, 1, 2, 15)	(4, 4, 5, 20)
4	(1, 1, 4, 15)	(2, 1, 4, 15)		(4, 1, 3, 30)
5	(1, 2, 5, 20)	(2, 1, 3, 20)		(4, 1, 5, 40)
6	(1, 1, 5, 25)			(4, 1, 2, 45)
7	(1, 3, 4, 25)			
8	(1, 3, 5, 35)			

For k=1, the problem simplifies to finding the transaction with the largest profit: (4, 1, 2, 45). However, for k=2, we cannot simply look at the next highest transaction, (4, 1, 5, 40), as it conflicts with our first choice. In situations where k>1, we must also consider the duration for which the stock was held. Transactions with a longer holding period could lead us to ignoring shorter transactions with a higher net profit. Therefore, when making a choice, we must carefully evaluate all other conflicting transactions at both the buy and sell stages. Since we have the freedom to trade multiple stocks in one day, we only need to examine transactions involving the same stock when addressing conflicts. One way to evaluate the trade offs could be to group complementary choices together.

Complementary choices for Stock 4 (k > 1)

$$\left| \begin{array}{c} (4,\ 1,\ 2,\ 45) \rightarrow (4,3,5,10) \\ (4,\ 1,\ 2,\ 45) \rightarrow (4,4,5,20) \end{array} \right|$$

Complementary choices for Stock 3 (k > 1)

$$\left| \begin{array}{c} (3,\ 1,\ 2,\ 15) \rightarrow (3,4,5,5) \\ (3,\ 1,\ 3,\ 5) \rightarrow (3,4,5,5) \end{array} \right|$$

Complementary choices for Stock 1 (k > 1)

$$\left| \begin{array}{c} (1,\ 1,\ 2,\ 45) \rightarrow (1,3,4,25) \\ (1,\ 1,\ 2,\ 45) \rightarrow (1,3,5,35) \\ (1,\ 1,\ 2,\ 45) \rightarrow (1,4,5,10) \end{array} \right|$$

Using the tables above we can easily determine the trade-off of a particular choice. For example, we can conclude that (1, 3, 5, 35) is the best choice after (4, 1, 2, 45) for k = 2, because (4, 1, 5, 40) can be ruled out, as it conflicts and has no benefit over (4, 1, 2, 45). If a sequence of choices within one stock has a higher net value than a current choice we know that it will be present as K increases.

K	Transactions	Profit
k_1	[(4, 1, 2)]	45
k_2	[(4, 1, 2) (1, 3, 5)]	80
k_3	[(4, 1, 2) (1, 3, 5) ((4, 4, 5, 20) OR (2, 1, 3, 20))]	100
k_4	[(4, 1, 2) (1, 3, 5) (4, 4, 5) (2, 1, 3)]	120
k_5	[(4, 1, 2) (1, 3, 5) (4, 4, 5) (2, 1, 3) (3, 1, 2)]	135
k_6	[(4, 1, 2) (1, 3, 5) (4, 4, 5) (2, 1, 3) (3, 1, 2) ((1, 1, 2) OR (3, 4, 5))]	140
k_7	[(4, 1, 2) (1, 3, 5) (4, 4, 5) (2, 1, 3) (3, 1, 2) (1, 1, 2) (3, 4, 5)]	145

8.2 Task 4. Dynamic Programming Solution To Problem 2

In this example we do a dynamic programming solution for problem 2. Here we implement an array that serves as a dynamic table with columns representing the number of days, and rows representing the number of transactions. In the array we stores an array with two indices, one with the maximum profit possible given i-1th days and j-1th transactions, and the second index contains another array with the transactions made for that profit. We subtract one from the index because we are interested in having a row/column for zero days and/or zero transactions. We iterate over the number of transactions and then the number of days we can sell. At each of these iterations we put the old best profit on our dynamic table and then look at each of the stocks and their sell days to see if we get a better profit. The time complexity for this algorithm is

$$O(stock * days^{2*k}$$

8.2.1 Assumption and Definitions

In this section we make the same assumptions as before for 1 indexing and the size() function which outputs the dimensions of a matrix. Additionally we define initializing a 2d array such that, array = [v1 * m] * n, which creates a 2d matrix with value v1. Finally we assume the behaviour of summing two arrays with the '+' operator will append them. For example a = [1, 2] and b = [3], if c = a + b, then c = [1, 2, 3]

8.2.2 Pseudo Code

procedure KTransactionsDynamic(Stock Matrix = A, Transaction Limit = k)
  solutionArray ← [0, 0, 0, 0]
  [rowDim, colDim] ← size(A)
  dynamicTable ← [[0, []] * colDim] * (k + 1)
  for transactions = 2 to k + 1 **do**
    for sellDay = 2 to colDim **do**
      previous ← dynamicTable[transactions][sellDay - 1]
      dynamicTable[transactions][sellDay] ← previous

for stockIndex = 1 to rowDim **do**
for buyDay = 1 to sellDay - 1 **do**
buyPrice ← A[stockIndex][buyDay]
sellPrice ← A[stockIndex][sellDay]
profit ← sellPrice - buyPrice
previousProfit ← dynamicTable[transactions - 1][buyDay][1]
previousTransactions ← dynamicTable[transactions - 1][buyDay][1]
netProfit ← profit + previousProfit
if netProfit > dynamicTable[transactions][sellDay][0] **then**
dynamicTable[transactions][sellDay][0] ← netProfit
solutionArray ← previousTransactions + [[stockIndex, buyDay, sellDay]]
dynamicTable[transaction][sellDay][1] ← solutionArray
end if
end for
end for
end for
end for
return dynamicTable[k][colDim][1]
end procedure

8.2.3 Python Implementation

```
import numpy as np

def k_transactions_dynamic(stock_matrix, k):
    solution_array = [0, 0, 0, 0]
    row_dim, col_dim = np.shape(stock_matrix)
    dynamic_table = [
    [[0, []] for _ in range(col_dim)] ...
    for _ in range(k + 1)]

    for transactions in range(2, k + 1):
        for sell_day in range(1, col_dim):
            previous = dynamic_table[transactions][sell_day - 1]
            dynamic_table[transactions][sell_day] = previous.copy()

            for stock_index in range(row_dim):
                for buy_day in range(sell_day):
                    buy_price = stock_matrix[stock_index][buy_day]
                    sell_price = stock_matrix[stock_index][sell_day]
                    profit = sell_price - buy_price
```

```
            previous_profit = dynamic_table[transactions ...
            - 1][buy_day][0]
            previous_transactions = dynamic_table[transactions ...
            - 1][buy_day][1]
            net_profit = profit + previous_profit

            if net_profit > ...
            dynamic_table[transactions][sell_day][0]:
                dynamic_table[transactions][sell_day][0] = ...
                net_profit
                dynamic_table[transactions][sell_day][1] = ...
                previous_transactions + [[stock_index, ...
                buy_day, sell_day]]

    return dynamic_table[k][col_dim - 1][1]
```

8.2.4 Description, Analysis, and Limitations, for Task 4

This algorithm uses a dynamic approach to calculate the best stocks to buy and sell given a number of transactions. In this case we have a dynamic table, composed of a 3D array, to store values to speed up computation. In the first layer of the 3D array we store the actual profit, and in the second layer we store a list of the transactions that are needed to make that profit.

9 Problem 3

9.1 Solution

$$A = \begin{bmatrix} 7 & 1 & 5 & 3 & 6 & 8 & 9 \\ 2 & 4 & 3 & 7 & 9 & 1 & 8 \\ 5 & 8 & 9 & 1 & 2 & 3 & 10 \\ 9 & 3 & 4 & 8 & 7 & 4 & 1 \\ 3 & 1 & 5 & 8 & 9 & 6 & 4 \end{bmatrix}$$

$$\text{c} = 2$$

We can calculate the maximum profit possible if a stock was purchased on a given day, given the constraint of not being able to sell until c + 1 days. We ignore any purchases done at $days \geq totaldays - (c+1)$

Stock Index \ Days	1	2	3	4
1	(1, 7, 2)	(2, 7, 8)	(3, 7, 4)	(4, 7, 6)
2	(1, 5, 7)	(2, 5, 5)	(3, 7, 5)	(4, 7, 1)
3	(1, 7, 5)	(2, 7, 2)	(3, 7, 1)	(4, 7, 9)
4	(1, 4, -1)	(2, 5, 4)	(3, 6, 0)	(4, 7, -7)
5	(1, 5, 6)	(2, 5, 8)	(3, 6, 1)	(4, 7, -4)
Maximum profit	7	8	5	9

Next, we can calculate the maximum profit after a stock purchase on a given day. Assuming we can sell a stock before cool down period c, having c only affect purchases.

Stock Index \ Days	1	2	3	4	5	6
1	2	8	4	6	3	1
2	7	5	6	2	-1	7
3	5	2	1	9	8	7
4	-1	5	4	-1	-3	-3
5	6	8	4	1	-3	-2
Maximum profit	7	8	6	9	8	7

Next we can think about how a sequence of transactions could be strung together given constraint c = 2. Looking at the maximum profit of a single transaction we can narrow down the maximum profit to be ≥ 9 from transaction (3, 4, 7, 9). Given constraint c, there is a very limited combinations of multiple transactions.

Stock Index \ Days	1	2	3	4	5	6
1	Buy	Sell	-	Buy	Sell	-
2	Buy	Sell	-	Buy	-	Sell
3	Buy	Sell	-	-	Buy	Sell
4	Buy	-	Sell	-	Buy	Sell
5	-	Buy	Sell	-	Buy	Sell

Finding the maximum profit of a transaction for each of the individual buy/sell timings shown above, we can start to combine transactions to achieve a value greater than 9.

Transaction Timing \ Days	1	2	3	4	5	6	Max Profit Transaction
1	Buy	Sell	-	-	-	-	(3, 1, 2, 3)
2	Buy	-	Sell	-	-	-	(5, 1, 3, 2)
3	-	Buy	Sell	-	-	-	(5, 2, 3, 4)
4	-	-	-	Buy	Sell	-	(1, 5, 6, 2)
5	-	-	-	Buy	-	Sell	(3, 5, 7, 8)
6	-	-	-	-	Buy	Sell	(2, 6, 7, 7), (3, 6, 7, 7)

Comparing the two previous tables we can determine the maximum profit from multiple transactions. The resulting profit, 11, is higher than the previously identified highest value of 9.

$$MaximumProfit = \big[(3,1,2,3)(3,5,7,8)\big]\ OR\ \big[(5,2,3,4)(2,6,7,7)\big]\ OR\ \big[(5,2,3,4)(3,6,7,7)\big]$$