Assignment 2

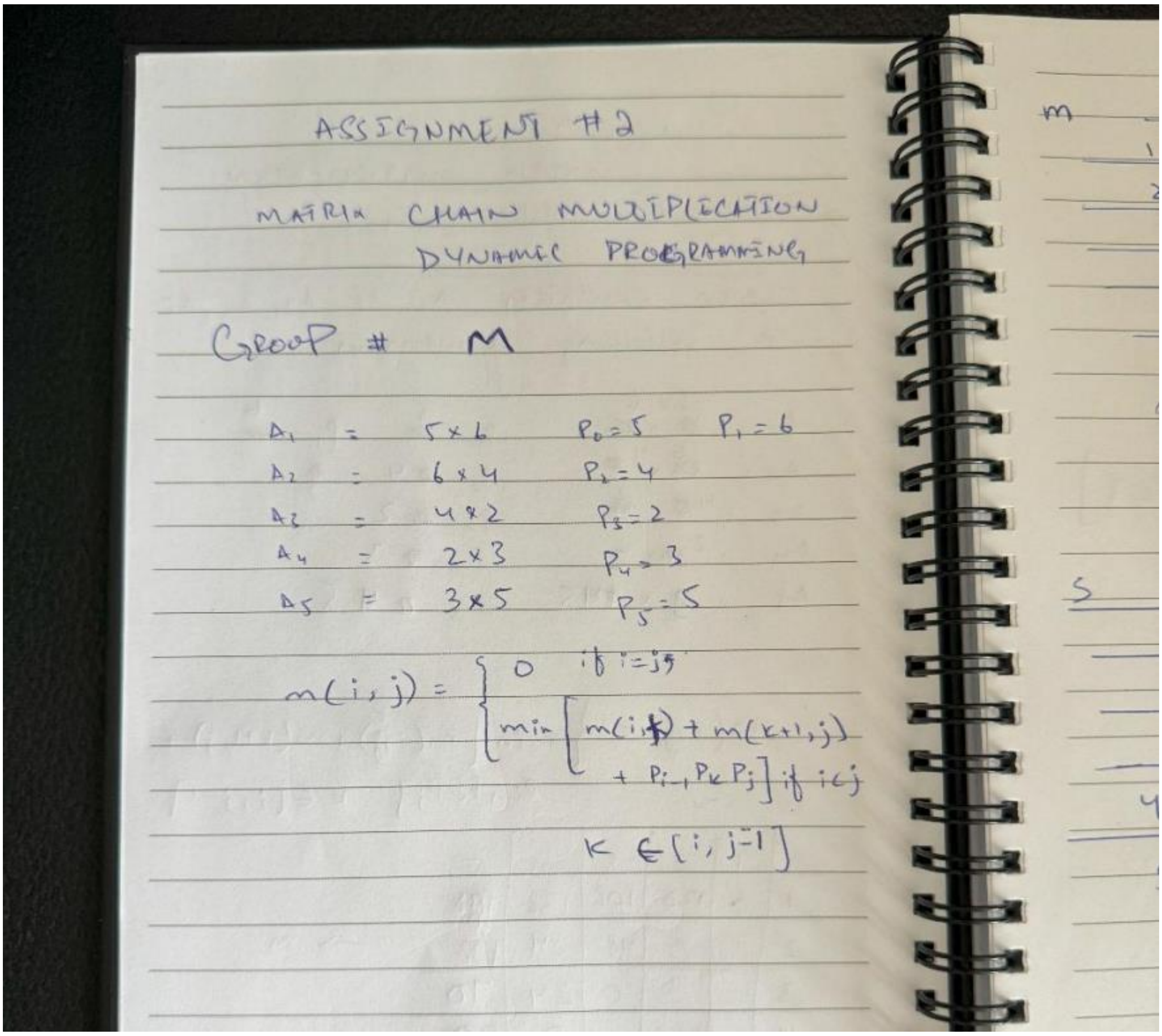
Group M

Jakub Mroz, Muhammad Fahad Nawaz Rana

Code is available at: https://github.com/JakubMroz4/algorithms

Part 1

1. Hand Written Solution:



ASSIGNMENT #2

MATRIX CHAIN MULTIPLICATION

DYNAMIC PROGRAMMING

GROUP # M

$A_1 = 5 \times 6$ $P_0 = 5$ $P_1 = 6$

$A_2 = 6 \times 4$ $P_2 = 4$

$A_3 = 4 \times 2$ $P_3 = 2$

$A_4 = 2 \times 3$ $P_4 = 3$

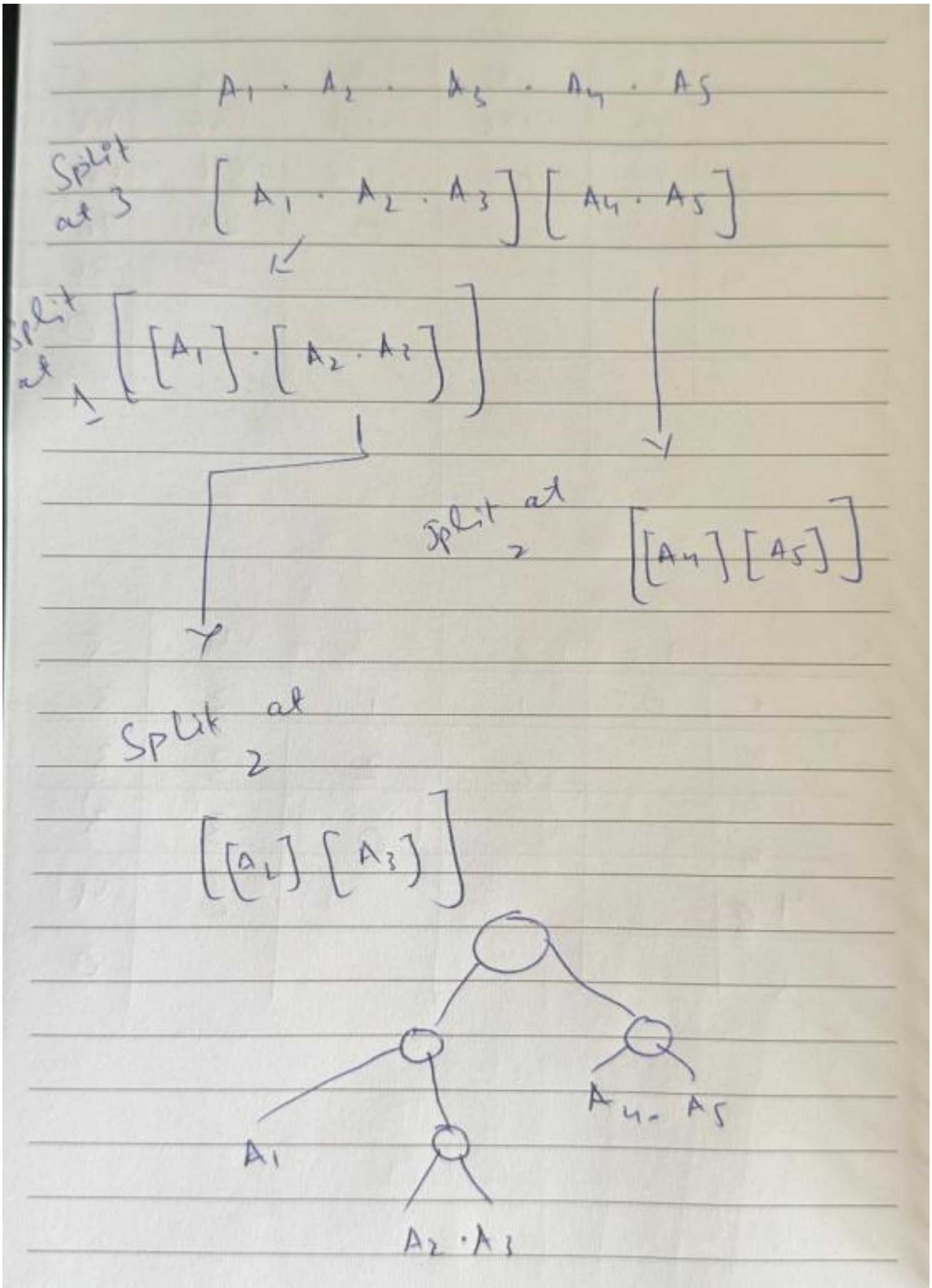
$A_5 = 3 \times 5$ $P_5 = 5$

$$m(i,j) = \begin{cases} 0 & \text{if } i=j \\ \min\left[m(i,k) + m(k+1,j) + P_{i-1} P_k P_j\right] & \text{if } i<j \end{cases}$$

$$k \in [i, j-1]$$

m	1	2	3	4	5
1	0	120	108	138	188
2		0	48	84	138
3			0	24	70
4				0	30
5					0

s	1	2	3	4	5
1	0	1	1	3	3
2		0	2	3	3
3			0	3	3
4				0	4
5					0



2. Implement a dynamic programming program that solves the problem for any number of matrices.

Output:

```
PROBLEMS    OUTPUT    DEBUG CONSOLE    TERMINAL    PORTS                                    Python

PS D:\Education\UiS\MS Computer Science\Spring 2024, 2nd\DAT 600, Algorithm Theory\Assignment 2> & C:/Python312/python.exe "d:/Education/UiS/MS Computer Science/Spri
ng 2024, 2nd/DAT 600, Algorithm Theory/Assignment 2/app.py"
Memoization table m:
[0, 120, 108, 138, 188]
[0, 0, 48, 84, 138]
[0, 0, 0, 24, 70]
[0, 0, 0, 0, 30]
[0, 0, 0, 0, 0]

Memoization table s:
[0, 1, 1, 3, 3]
[0, 0, 2, 3, 3]
[0, 0, 0, 3, 3]
[0, 0, 0, 0, 4]
[0, 0, 0, 0, 0]
PS D:\Education\UiS\MS Computer Science\Spring 2024, 2nd\DAT 600, Algorithm Theory\Assignment 2>
```

Code:

```python
def matrix_chain_order(p):
    # p is a list of matrix dimensions, where p[i-1] x p[i] is the dimension of the i-th matrix
    n = len(p) - 1  # number of matrices
    m = [[0] * (n + 1) for _ in range(n + 1)]  # memoization table for minimum scalar multiplications
    s = [[0] * (n + 1) for _ in range(n + 1)]  # memoization table for optimal split points

    # Initialize the diagonal of m to 0
    for i in range(1, n + 1):
        m[i][i] = 0

    # Fill the memoization tables
    for l in range(2, n + 1):  # l is the chain length
        for i in range(1, n - l + 2):
            j = i + l - 1  # End index of the chain
            m[i][j] = float('inf')  # Initialize to infinity
            for k in range(i, j):  # Try all possible split points
                q = m[i][k] + m[k + 1][j] + p[i - 1] * p[k] * p[j]  # Compute cost
                if q < m[i][j]:  # If the cost is less than the current minimum
                    m[i][j] = q  # Update the minimum cost
                    s[i][j] = k  # Update the optimal split point

    # Print memoization tables m and s
    print("Memoization table m:")
    for row in m[1:]:
        print(row[1:])

    print("\nMemoization table s:")
    for row in s[1:]:
        print(row[1:])

    return m, s


# Example dimensions of matrices A1, A2, A3, A4, A5
dimensions = [5, 6, 4, 2, 3, 5]

m, s = matrix_chain_order(dimensions)
```

3. Does it exist a greedy choice that could apply to this problem?

Yes, there exists a greedy choice that can be applied to the matrix chain multiplication problem. Utilizing a heuristic that frequently minimises scalar multiplications, the 'greedy matrix chain multiplication' method selects the next pair of matrices to multiply. Because the issue lacks the greedy-choice feature, locally optimal options do not necessarily result in a globally optimal solution, hence it does not guarantee optimal solutions. An increasingly dependable method for locating the ideal solution is dynamic programming

Part 2

In a knapsack problem, we get a list of values and weights for items, and a total knapsack capacity. The problem handles packing the knapsack with the highest value possible, without exceeding the weight capacity.

In 01 knapsack problem, we can only fit the entire item or leave it. In fractional Knapsack we can take a part of an item.

Fractional knapsack problem has an easy greedy solution – take the object with the highest value / weight ratio and try to fit it completely. If it cannot be fitted, fit as much of it as possible. Repeat this until the knapsack is full or there is no more items.

01 knapsack is harder, as taking items with highest value / weight ratio will not always be the right solution.

We follow the pseudocode from wiki describing the problem:
https://en.wikipedia.org/wiki/Knapsack_problem#0-1_knapsack_problem

First we initialize a matrix (len(items) + 1 X capacity + 1)

Then we create two for loops, first iterates over len(items) + 1,

and second iterates over capacity + 1

To find the maximum value for each cell in the matrix[item][weight] , we have to choose the bigger (max) result of two possibilities:

1. Include the n item (value is n + previous n-1 items)

2. Don't include the item if its weight exceeds the capacity (value remains the same as in the previous n – 1 items)

Code for fractional knapsack program:

```
4 usages  JakubMroz4
class Item:
     JakubMroz4
    def __init__(self, value, weight):
        self.value = value
        self.weight = weight

    1 usage (1 dynamic)  JakubMroz4
    def ratio(self):
        return self.value / self.weight


2 usages  JakubMroz4
def fractional_knapsack(items: list, capacity: int):
    # sort by value / weight ratio, decreasingly
    items.sort(key=lambda x: x.ratio(), reverse=True)
    max_value = 0.0
    solution_items = []

    if capacity == 0:
        return 0, list()

    # iterate over items
    for index, item in enumerate(items):

        # try to add entire item
        if item.weight <= capacity:
            capacity -= item.weight
            max_value += item.value
            solution_items.append(index + 1)

        # else add fraction of it
        else:
            max_value += item.value * capacity / item.weight
            solution_items.append(index + 1)
            break

    return max_value, solution_items

```

Code for 01 knapsack program:

```python
2 usages  JakubMroz4
def knapsack_01(weights: list, values: list, capacity: int):
    length = len(weights)

    # create 2d array Capacity + 1 x Items + 1
    # last value in matrix[i] will be max value for the last i items
    matrix = [[0 for _ in range(capacity + 1)] for _ in range(length + 1)]

    # iterate over items
    for i in range(1, length + 1):
        # iterate over weights
        for w in range(1, capacity + 1):

            # if next items does not exceed capacity
            # choose max of not including the item or including the item
            if weights[i - 1] <= w:
                matrix[i][w] = max(matrix[i - 1][w], values[i - 1] + matrix[i - 1][w - weights[i - 1]])

            # if capacity exceeded: do not include the item
            else:
                matrix[i][w] = matrix[i - 1][w]

    max_value = matrix[length][capacity]
    return max_value, matrix
```

The program main program gets the weights and values in 2 lists, and shuffles them for every problem

```python
def knapsack_program(weights: list, values: list, capacity: int, amount: int):
    print("Item indexes start with 1")

    for x in range(amount):
        shuffle(weights)
        shuffle(values)

        print(f"\nProblem #{x+1}")
        print(f"Values: {values}")
        print(f"Weights: {weights}")

        k01.input_wrapper(weights, values, capacity)
        kf.input_wrapper(weights, values, capacity)
```

Output given by the program:

```
Item indexes start with 1
Capacity: 100

Problem #1
Values: [240, 120, 55, 10, 200, 72]
Weights: [20, 80, 35, 60, 10, 40]

01 Knapsack
Solution: 512
Items in the solution: [1, 5, 6]

Fractional Knapsack
Solution: 559.1428571428571
Items in the solution: [5, 1, 6, 3]

Problem #2
Values: [10, 240, 120, 200, 72, 55]
Weights: [60, 40, 20, 35, 10, 80]

01 Knapsack
Solution: 560
Items in the solution: [2, 3, 4]

Fractional Knapsack
Solution: 603.4285714285714
Items in the solution: [5, 2, 3, 4]
```

Part 3

Given an array of coins $c1 < c2 < . . . < c_n$, the objective is to determine the fewest coins needed to achieve a total of N .

1. Propose a greedy solution to minimize the number of coins required for a given total N .

initialize variable total = 0

In a loop: try to add the biggest coin possible to total, such that total <= N

Exit the loop when total == N

2. Some currency systems may pose challenges for a greedy approach. For *instance, with coins = [1, 5, 11], a greedy solution for N = 15 will not yield*

the minimum number of coins 11 + 1 + 1 + 1 + 1 = 15 is 5 coins but an

optimal solution would be 5 + 5 + 5 = 15 which is 3 coins.

3. Devise a new solution that accommodates any currency system, ensuring

an optimal global solution for the minimum number of coins required:

We can solve this problem in a similar way to 01 knapsack problem, the only difference being we can use an "item" multiple times.

Lets start with initializing an array of len(N + 1).

Then lets iterate over the cells in the array and fill in the minimum amount of coins for each cell (index of the cell is equal to temporary N in the calculations)

In each cell, for each coin evaluate if adding the coin is better than the previous result, and if makes the sum greater than N.

Fill the matrix this way

4. Find out if the Norwegian coin system is greedy, coins = [1, 5, 10, 20].

Norwegian coin system is greedy. In the example given in the task above with coins = [1, 5, 11], the problem is that 11 is not a multiple of a lower value coin 5 and it cannot be used as a substitute for multiple of coins with value 5.

This is not a problem in the Norwegian system, as 20 is a multiple of 10, and 10 is a multiple of 5. Therefore, any multiple of 5s can be substituted into higher value coins in a greedy algorithm, which always tries to fit the largest coin possible.

5. What is the running time of these two algorithms?

Dynamic programming algorithm which fits every coin combination has a running time O(n)

Greedy approach has a time complexity O(n log(n))