Assignment 2. Dynamic and Greedy Algorithms.

Student name: Evgenii Dudkin
e-mail: e.dudkin@stud.uis.no

1 Task 1. Matrix Chain Multiplication

The problem at hand is to solve the matrix chain multiplication problem, specifically with five matrices. Table 1 illustrates the dimensions of these five matrices.

matrix	A1	A2	A3	A4	A5
dimension	10x5	5x50	50x2	2x100	100x10

Table 1: Matrices with their respective dimensions

1.1

While manually solving the parenthesization problem using the recursive formula, we populate the tables m and s with their respective values. At index i x j table m stores the minimum cost for multiplying $A_i ... A_j$ and table s records the index k that attained the optimal cost when computing m[i,j].

After populating tables m and s, they are depicted in Figure 1. The solution, which determines the optimal placement of parentheses to minimize the number of operations during the matrix chain multiplication, is represented as $(A_1(A_2A_3))(A_4A_5)$, with a minimal cost of 2800 for such multiplication.

1.2

A dynamic programming algorithm was implemented in C# to solve the matrix chain multiplication problem for any number of matrices. This library allows users to input an array of matrix dimensions and receive a solution with parentheses indicating the optimal grouping, along with the associated minimal cost. The actual implementation employs the same recursive formula approach, utilizing auxiliary tables m and s. Figure 2 illustrates the output of the program using the same data as presented in Table 1. It is evident that the solution obtained manually and via the programming solution are identical, affirming the correctness of the dynamic programming algorithm.

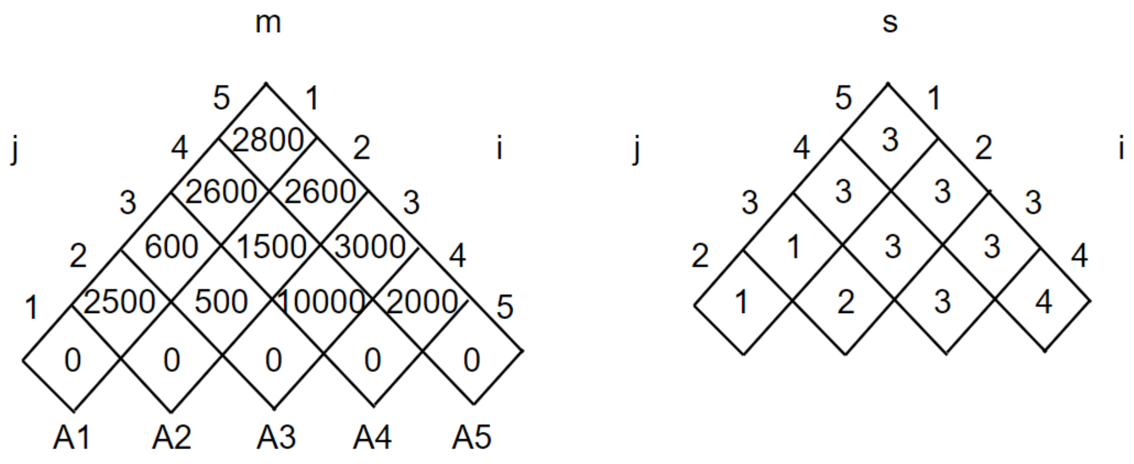
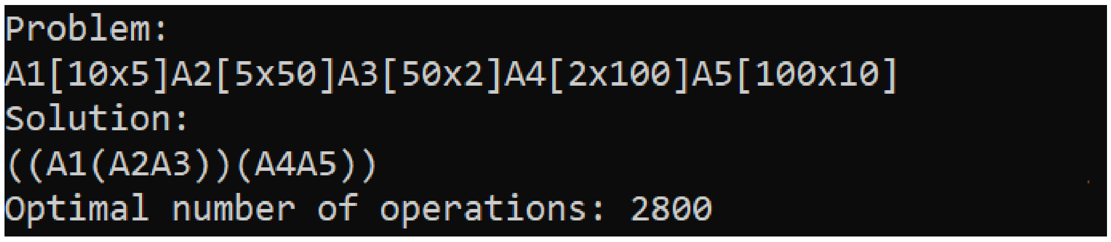


Figure 1: The m and s tables

For access to the implementations, code, comments, and other related materials, please refer to the GitHub repository [1].

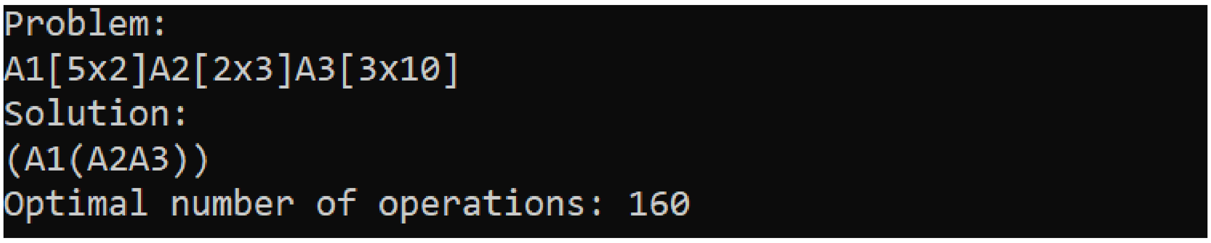


```
Problem:
A1[10x5]A2[5x50]A3[50x2]A4[2x100]A5[100x10]
Solution:
((A1(A2A3))(A4A5))
Optimal number of operations: 2800
```

Figure 2: Matrix chain multiplication program's output

1.3

The presence of a feasible greedy approach for this problem is nonexistent. No available greedy strategies can guarantee a correct outcome. One potential strategy might involve performing multiplications in sequence. For instance, with three matrices $A_1A_2A_3$, a possible solution would be $(A_1A_2)A_3$. However, it becomes apparent that this method will not yield accurate results. Consider the matrices A_1 (5x2), A_2 (2x3), and A_3 (3x10). The greedy solution yields 180 operations as optimal, while the actual optimal solution would be $(A_1(A_2A_3))$ with 160 operations. This can be confirmed by the dynamic programming algorithm discussed earlier (Figure 3).



```
Problem:
A1[5x2]A2[2x3]A3[3x10]
Solution:
(A1(A2A3))
Optimal number of operations: 160
```

Figure 3: Greedy solution counterargument

2 Task 2. Fractional and 0-1 Knapsack

A C# program was developed to address both the 0-1 knapsack problem and the fractional knapsack problem, taking into account the potential units (e.g., kg, price) and the maximum capacity of the knapsack as input parameters. Figure 4 provides an illustration of the program's actual execution.

2.1

The 0-1 knapsack problem is tackled through a dynamic programming approach, iteratively populating the matrix m with a size of nxM (where n represents the number of items and M denotes the maximum capacity of the knapsack). Here, m[i,j] retains the maximum profit achieved by considering items from 1 to i, while ensuring the total weight does not exceed j.

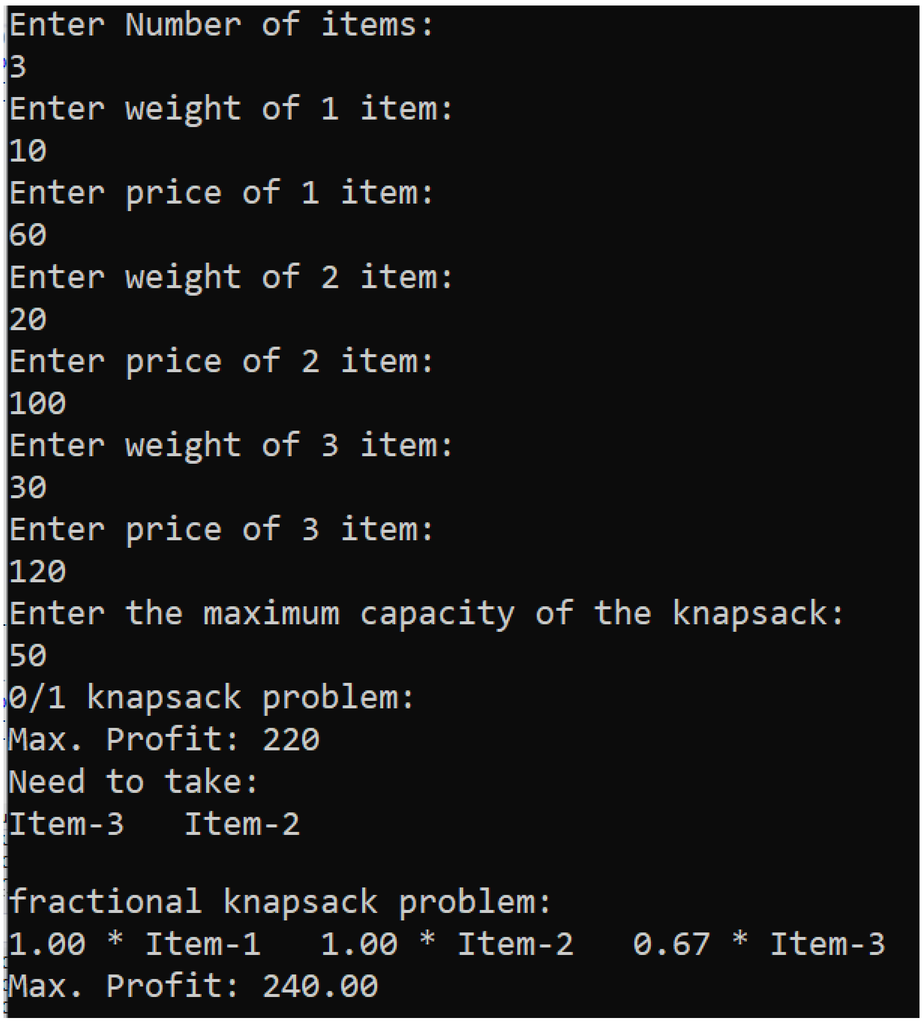
2.2

The fractional knapsack problem is addressed through a greedy approach. An auxiliary list of items, encompassing each item alongside its price/weight ratio to describe its profitability, is created and subsequently sorted in descending order. During the iteration through this sorted list, an item is selected in its entirety if the knapsack's capacity allows, or partially selected if it's too large to be accommodated entirely. Alternatively, if the knapsack is already full, the item is skipped.

3 Task 3. Coin change

3.1

The suggested greedy solution for minimizing the number of coins required to achieve a given total N, based on a provided array of coins, involves iterating from



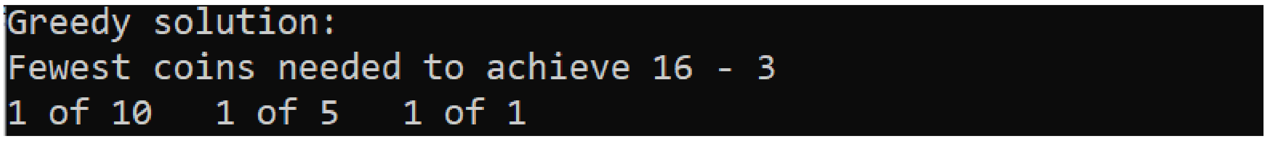
```
Enter Number of items:
3
Enter weight of 1 item:
10
Enter price of 1 item:
60
Enter weight of 2 item:
20
Enter price of 2 item:
100
Enter weight of 3 item:
30
Enter price of 3 item:
120
Enter the maximum capacity of the knapsack:
50
0/1 knapsack problem:
Max. Profit: 220
Need to take:
Item-3   Item-2

fractional knapsack problem:
1.00 * Item-1   1.00 * Item-2   0.67 * Item-3
Max. Profit: 240.00
```

Figure 4: Knapsack execution

the most valuable coin to the least. If the coin's denomination is higher than the remaining balance, the coin is skipped. Otherwise, the maximum number of coins that do not exceed the remaining balance is added to the solution. Subsequently, the balance is decreased by the total amount achieved, and the process continues iteratively.

Figure 5 illustrates the functionality of the implemented greedy strategy using coins [1, 5, 10] to attain a required total of 16. In this scenario, the solution would be $10 + 5 + 1 = 16$ (comprising 3 coins), a result validated by the algorithm.

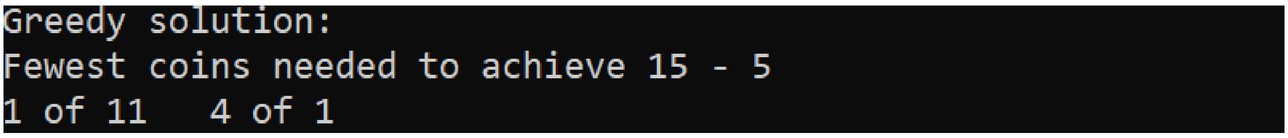


```
Greedy solution:
Fewest coins needed to achieve 16 - 3
1 of 10   1 of 5   1 of 1
```

Figure 5: Greedy solution execution

3.2

Certain currency systems can present difficulties for a greedy approach. Figure 6 illustrates an incorrect solution provided by the greedy algorithm when using the coins [1, 5, 11] to achieve a total of 15. The optimal solution would involve 3 coins (5 + 5 + 5), rather than the 5 coins (11 + 1 + 1 + 1 + 1) suggested by the program.



```
Greedy solution:
Fewest coins needed to achieve 15 - 5
1 of 11   4 of 1
```

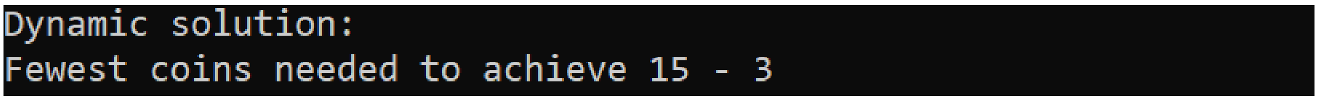
Figure 6: Greedy solution counterargument

3.3

A new solution, designed to accommodate any currency system and ensure an optimal global solution for the minimum number of coins required, has been implemented using a dynamic programming approach. In this implementation, an array of size N+1 is created, intended to store the minimal number of coins needed to achieve a total of i at the i-th position. This array (m) is initialized as follows: m[0]=0 and m[1..N]=∞.

The algorithm operates by iterating through the array m from 1 to N. Within the loop, all coins are iterated through, calculating the minimal number of coins required to achieve the current value indicated by the iterator. This calculation is performed by comparing m[i−coinRate]+1 and m[i], provided that m[i−coinRate] is not equal to infinity.

Figure 7 demonstrates the functionality of the dynamic approach using the coins [1, 5, 11] to achieve a total of 15. Consequently, a correct result of 3 coins (5 + 5 + 5) is obtained.



```
Dynamic solution:
Fewest coins needed to achieve 15 - 3
```

Figure 7: Dynamic solution execution

3.4

The standard condition for the greedy algorithm to be effective is that each coin denomination is divisible by the previous one. Thus, Norwegian coin system ([1,5,10,20]) is greedy.

3.5

The running time for the greedy algorithm is O(n), where n is the number of coins in the array. The running time for the dynamic programming algorithm is O(n*N), where N is the total needed to achieve.

References

[1] Github Evgenii Dudkin. — URL: `https://github.com/EvgeneDudkin/dat600` (online; accessed: 11.02.2024).