# Assignment 2

Group M

Jakub Mroz, Muhammad Fahad Nawaz Rana

Code is available at:

# Part 1

1. Hand Written Solution:

Obraz zawierający tekst, notatnik, pismo odręczne, Materiały biurowe

Opis wygenerowany automatycznie

Obraz zawierający tekst, pismo odręczne, dokument

Opis wygenerowany automatycznie

Obraz zawierający tekst, pismo odręczne, notatnik, papier

Opis wygenerowany automatycznie

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

Output:

Obraz zawierający tekst, oprogramowanie, Oprogramowanie multimedialne, zrzut ekranu

Opis wygenerowany automatycznie

Code: Obraz zawierający tekst, zrzut ekranu, oprogramowanie

Opis wygenerowany automatycznie

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. The greedy approach aims to choose the next pair of matrices to multiply based on a certain heuristic, typically aiming to minimize the number of scalar multiplications. One common heuristic is to choose the pair of matrices that minimizes the number of rows and columns in the resulting intermediate matrix.

However, while the greedy approach might work in some cases, it does not guarantee an optimal solution for all instances of the matrix chain multiplication problem. In fact, the greedy approach can lead to suboptimal solutions in many cases.

The reason for this is that the matrix chain multiplication problem exhibits optimal substructure but not necessarily the greedy-choice property. Optimal substructure means that an optimal solution to the problem contains optimal solutions to its subproblems. However, the greedy-choice property requires that a globally optimal solution can be arrived at by making a locally optimal choice at each step, which is not always true for the matrix chain multiplication problem.

Therefore, while a greedy algorithm can be devised for the matrix chain multiplication problem, it may not always produce the optimal solution. Dynamic programming, as implemented in the previous step, is a more reliable approach to finding the optimal solution for this problem

# 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:

Obraz zawierający tekst, zrzut ekranu, oprogramowanie, Oprogramowanie multimedialne

Opis wygenerowany automatycznie

Code for 01 knapsack program:

Obraz zawierający tekst, zrzut ekranu

Opis wygenerowany automatycznie

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

Obraz zawierający tekst, zrzut ekranu, oprogramowanie, Czcionka

Opis wygenerowany automatycznie

Output given by the program:

Obraz zawierający tekst, zrzut ekranu, Czcionka, menu

Opis wygenerowany automatycznie

# 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))