DAT600: Algorithm Theory - Assignment 2

Antón Maestre Gómez - Daniel Linfon Ye Liu

Task 1 Problem Matrix Chain Multiplication:

Matrix Chain Multiplication is an optimization problem that determines the most efficient way to parenthesize a sequence of matrix multiplications to minimize the total number of scalar multiplications. The algorithm follows a dynamic programming approach using two matrices:

- `M`: Stores the minimum cost of multiplying matrices within a given range.
- `S`: Stores the optimal split points to reconstruct the parenthesization.

Let's take the following matrices as an example:

Matrix	A1	A2	A3	A4	A5
Dimension	10x20	20x30	30x40	40x50	50x60

First, we are going to solve the parenthesization problem by hand using the recursive formula to fill the memorization tables m and s.

To solve the parenthesization problem and fill the tables m and s:

$$m[i,j] = \begin{cases} 0 & \text{if } i = j \\ \min_{i \le k < j} \{ m[i,k] + m[k+1, j] + p_{i-1} \cdot p_k \cdot p_j \} & \text{if } i < j \end{cases}$$

1st level

$m[1,1], m[2,2], m[3,3], m[4,4], m[5,5] = 0$

2nd level

$m[1,2] = \min_{k=1} \{ m[1,1]^0 + m[2,2]^0 + 10 \cdot 20 \cdot 30 = 6000 \}$ ($p_0\ p_1\ p_2$)

$s[1,2] = 1$

$m[2,3] = \min_{k=2} \{ m[2,2]^0 + m[3,3]^0 + 20 \cdot 30 \cdot 40 = 24000 \}$ ($p_1\ p_2\ p_3$)

$s[2,3] = 2$

$m[3,4] = \min_{k=3} \{ m[3,3]^0 + m[4,4]^0 + 30 \cdot 40 \cdot 50 = 60000 \}$ ($p_2\ p_3\ p_4$)

$s[3,4] = 3$

$m[4,5] = \min_{k=4} \{ m[4,4]^0 + m[5,5]^0 + 40 \cdot 50 \cdot 60 = 120000 \}$ ($p_3\ p_4\ p_5$)

$s[4,5] = 4$

3rd level

$m[1,3] = \min \begin{cases} m[1,1]^0 + m[2,3]^{24000} + 10 \cdot 20 \cdot 40 = 32000 & (p_0\ p_1\ p_3),\ k=1 \\ m[1,2]^{6000} + m[3,3]^0 + 10 \cdot 30 \cdot 40 = 18000 & (p_0\ p_2\ p_3),\ k=2 \end{cases}$

$s[1,3] = 2$

$m[2,4] = \min \begin{cases} m[2,2] + m[3,4] + 20 \cdot 30 \cdot 50 = 90000 \\ k=2 \qquad 60000 \qquad P_1 P_2 P_4 \\ m[2,3] + m[4,4] + 20 \cdot 40 \cdot 50 = 64000 \\ k=3 \qquad 24000 \qquad 0 \qquad P_1 P_3 P_4 \end{cases}$

$S[2,4] = 3$

$m[3,5] = \min \begin{cases} m[3,3] + m[4,5] + 30 \cdot 40 \cdot 60 = 192000 \\ x=3 \qquad 0 \qquad 120000 \\ m[3,4] + m[5,5] + 30 \cdot 50 \cdot 60 = 150000 \\ k=4 \qquad 60000 \qquad 0 \end{cases}$

$S[3,5] = 4$

4: level

$m[1,4] = \min \begin{cases} m[1,1] + m[2,4] + 10 \cdot 20 \cdot 50 = 74000 \\ k=1 \qquad 0 \qquad 64000 \\ m[1,2] + m[3,4] + 10 \cdot 30 \cdot 50 = 81000 \\ k=2 \qquad 6000 \qquad 60000 \\ m[1,3] + m[4,4] + 10 \cdot 40 \cdot 50 = 38000 \\ k=3 \qquad 18000 \qquad 0 \end{cases}$

$S[1,4] = 3$

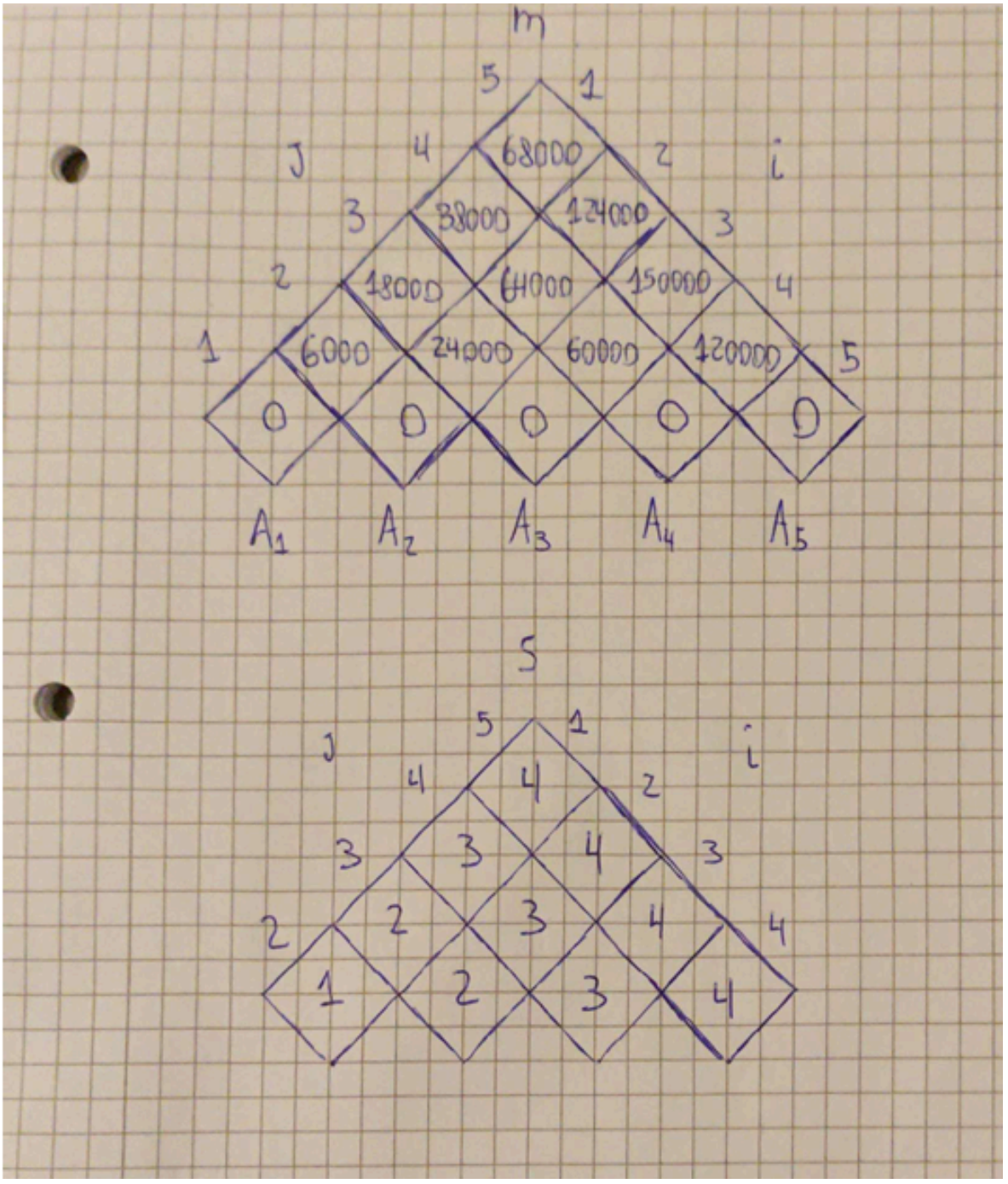
$m[2,5] = \min \begin{cases} m[2,2] + m[3,5] + 20 \cdot 30 \cdot 60 = 186000 \\ k=2 \qquad 0 \qquad 150000 \\ m[2,3] + m[4,5] + 20 \cdot 40 \cdot 60 = 192000 \\ k=3 \qquad 24000 \qquad 120000 \\ m[2,4] + m[5,5] + 20 \cdot 50 \cdot 60 = 124000 \\ k=4 \qquad 64000 \qquad 0 \end{cases}$

$S[2,5] = 4$

5: level

$m[1,5] = \min \begin{cases} m[1,1] + m[2,5] + 10 \cdot 20 \cdot 60 = 136000 \\ k=1 \qquad 0 \qquad 124000 \\ m[1,2] + m[3,5] + 10 \cdot 30 \cdot 60 = 174000 \\ k=2 \qquad 6000 \qquad 150000 \\ m[1,3] + m[4,5] + 10 \cdot 40 \cdot 60 = 162000 \\ k=3 \qquad 18000 \qquad 120000 \\ m[1,4] + m[5,5] + 10 \cdot 50 \cdot 60 = 68000 \\ k=4 \qquad 38000 \qquad 0 \end{cases}$

$S[1,5] = 4$


m
5
1
J
4
68000
2
i
3
38000
124000
3
2
18000
64000
150000
4
1
6000
24000
60000
120000
5
0
0
0
0
0
A1
A2
A3
A4
A5

5
5
1
J
i
4
4
2
3
3
4
3
2
2
3
4
4
1
2
3
4

Now, we have implemented a program that solves the problem for any number of matrices using the following functions:

```python
def matrix_chain_order(p):
    n = len(p) - 1
    m = [[0] * (n - i) for i in range(n)]
    s = [[0] * (n - i - 1) for i in range(n)]

    for l in range(2, n + 1):  # l is chain length
        for i in range(n - l + 1):
            j = i + l - 1
            m[i][j - i] = sys.maxsize
            for k in range(i, j):
                q = m[i][k - i] + m[k + 1][j - (k + 1)] + p[i] * p[k + 1] * p[j + 1]
                if q < m[i][j - i]:
                    m[i][j - i] = q
                    s[i][j - i - 1] = k + 1

    return m, s
```

```python
def print_optimal_parens(s, i, j):
    if i == j:
        print(f"A{i+1}", end="")
    else:
        print("(", end="")
        print_optimal_parens(s, i, s[i][j - i - 1] - 1)
        print_optimal_parens(s, s[i][j - i - 1], j)
        print(")", end="")
```

The function `matrix_chain_order(p)` computes the matrices using a bottom-up approach. and the function `print_optimal_parens(s, i, j)` reconstructs the optimal order of multiplication.

If we run the program with the data from the previous example, we get the same results calculated by hand.

```
Matrix M (minimum costs):
        j=1     j=2     j=3     j=4     j=5
i=1:      0    6000   18000   38000   68000
i=2:      0   24000   64000  124000
i=3:      0   60000  150000
i=4:      0  120000
i=5:      0

Matrix S (split points):
        j=2     j=3     j=4     j=5
i=1:      1       2       3       4
i=2:      2       3       4
i=3:      3       4
i=4:      4
i=5:
```

Calling `PRINT-OPTIMAL-PARENS(s,1,5)`, we obtain that the optimal parenthesization is:

((((A1A2)A3)A4)A5)

Is there a Greedy Strategy for Matrix Chain Multiplication?

The idea behind a greedy approach is to make optimal local decisions in the hope of obtaining a globally optimal result. In this case, a greedy strategy could be:

1. Always multiply the smallest matrices first.
2. Always multiply the matrices with the lowest immediate cost first.

Suppose we have three matrices with dimensions:

A1(10×100), A2(100×5), A3(5×50)

If we first multiply the matrices with the least amount of immediate operations:

A2×A3 = (100×5)×(5×50) = 100×5×50 = 25,000 operations.

Then, A1×(A2A3) = (10×100)×(100×50) = 10×100×50 = 50,000 operations.

Total: **75,000 operations.**

While the optimal solution with dynamic programming:

(A1×A2)×A3 = 5,000 operations

Then, (A1A2)×A3 = 2,500 operations

Total: **7,500 operations.**

The Matrix String Multiplication problem does **not have a greedy solution** because the minimum multiplication cost depends on the global structure of the sequence, not just on local decisions. To obtain the best solution, dynamic programming must be used.

Task 2 Fractional and 0-1 Knapsack

In this problem, we address two variants of the classic **Knapsack Problem**:

1. **Fractional Knapsack**: Allows taking fractions of items to maximize the total value.
2. **0-1 Knapsack**: Only allows including entire items (either you take it or leave it).

Both problems aim to maximize the total value of the items placed in the knapsack without exceeding its capacity. However, they differ in their constraints and solution strategies. The fractional version is solved using a **greedy algorithm**, while the 0-1 version requires **dynamic programming** to find the optimal combination of items.

- Input Data Generation

The `get_user_input()` function allows users to provide custom input for the knapsack problem. It begins by asking for the number of items to include. Then, through a loop, it prompts the user to enter the weight and value of each item individually, ensuring flexibility in defining the problem parameters.

After collecting the item details, the function requests the knapsack's total capacity, representing the maximum weight the knapsack can hold. Finally, it returns the lists of weights, values, and the capacity, enabling dynamic problem generation based on user input. This interactive approach enhances the program's usability and adaptability.

```python
def get_user_input():
    n_items = int(input("Enter the number of items: "))
    weights = []
    values = []
    os.system('cls' if os.name == 'nt' else 'clear')

    for i in range(n_items):
        weight = int(input(f"Enter the weight of item {i + 1}: "))
        value = int(input(f"Enter the value of item {i + 1}: "))
        os.system('cls' if os.name == 'nt' else 'clear')
        weights.append(weight)
        values.append(value)

    capacity = int(input("Enter the knapsack capacity: "))
    os.system('cls' if os.name == 'nt' else 'clear')

    return weights, values, capacity

# Get the data
weights, values, capacity = get_user_input()
```

- Fractional Knapsack Algorithm

The Fractional Knapsack algorithm employs a greedy approach. It begins by calculating the **value-to-weight ratio** for each item, which represents how much value each unit of weight contributes. Items are then sorted in descending order based on this ratio. The algorithm picks the item with the highest ratio first and continues until the knapsack's capacity is full.

If an item's full weight cannot be accommodated, the algorithm takes the fraction of that item that fits the remaining capacity. This ensures that the knapsack's capacity is used as efficiently as possible, often resulting in higher total values compared to the 0-1 version. The output consists of both the **maximum achievable value** and the **fractions of each item** taken to reach that value.

```python
def fractional_knapsack(weights, values, capacity):
    # Value-to-weight ratio calculation
    n = len(weights)
    value_per_weight = [(values[i] / weights[i], i) for i in range(n)]
    value_per_weight.sort(reverse=True)

    # Selection of items
    total_value = 0
    fractions = [0] * n
    for vpw, i in value_per_weight:
        # The item fits completely
        if capacity >= weights[i]:
            capacity -= weights[i]
            total_value += values[i]
            fractions[i] = 1
        # Only a fraction of the item fits
        else:
            fraction = capacity / weights[i]
            total_value += values[i] * fraction
            fractions[i] = fraction
            break

    return total_value, fractions
```

- 0-1 Knapsack Algorithm

The **0-1 Knapsack** problem is solved using **dynamic programming**. A two-dimensional table `dp` is constructed, where each entry `dp[i][w]` stores the maximum value obtainable using the first `i` items with a knapsack capacity of `w`.

For each item, two possibilities are considered:

- **Include the item**: If the current item's weight fits within the capacity, the algorithm compares the value of including the item against the value of excluding it.
- **Exclude the item**: If including the item exceeds the capacity, the maximum value from the previous item set is retained.

After filling the table, a **traceback** process identifies which items were included in the optimal solution. This output consists of the **maximum total value** and a list indicating whether each item was included (`1`) or excluded (`0`).

```python
def zero_one_knapsack(weights, values, capacity):
    n = len(weights)
    dp = [[0] * (capacity + 1) for _ in range(n + 1)]

    # Selection of items
    for i in range(1, n + 1):
        for w in range(capacity + 1):
            if weights[i-1] <= w:
                dp[i][w] = max(dp[i-1][w], dp[i-1][w - weights[i-1]] + values[i-1])
            else:
                dp[i][w] = dp[i-1][w]

    # Traceback to find included items
    w = capacity
    selected_items = [0] * n
    for i in range(n, 0, -1):
        if dp[i][w] != dp[i-1][w]:
            selected_items[i-1] = 1
            w -= weights[i-1]

    return dp[n][capacity], selected_items
```

For example, we have this input:

- **Weights** = [10, 20, 30]
- **Values** = [60, 100, 120]
- **Capacity** = 50

Each cell `dp[i][w]` represents the maximum value that can be obtained by considering the first `i` elements and a capacity `w`. The table is constructed as follows:

Item(i) \ Capacity(w)	0	10	20	30	40	50
0	0	0	0	0	0	0
1 (weight = 10)	0	60	60	60	60	60
2 (weight = 20)	0	60	100	160	160	160
3 (weight = 30)	0	60	100	160	180	220

The dynamic programming table starts with a **base case** (`i=0`), where no items are included, and every cell has a value of `0`. This forms the foundation for the solution.

- For the **first item** (weight = 10, value = 60), capacities of 10 or more allow including the item, setting the value to 60. Below this capacity, the value remains 0.

- The **second item** (weight = 20, value = 100) is considered next. For capacities ≥ 20, the algorithm checks whether including the item increases the total value compared to excluding it. For example, at a capacity of 30, including both the first and second items results in a total value of 160.
- The **third item** (weight = 30, value = 120) is evaluated similarly. At a capacity of 50, including the first and third items gives the best value of 220, as including the second and third together would exceed the capacity.

To identify the selected items, the algorithm starts from the final cell (`dp[3][50] = 220`) and compares it with previous rows. If the value differs, the item was included. Subtracting its weight updates the capacity until it reaches zero. The selected items are `[1, 1, 0]`, meaning the first and second items were included.

- Result Comparison

The two algorithms are compared based on the results they yield:

- The **Fractional Knapsack** typically produces a higher value since it allows splitting items into fractions.
- The **0-1 Knapsack** is more restrictive but guarantees an optimal combination with whole items only.

An example output looks like this:

```
Generated weights: [20, 40, 30]
Generated values: [100, 80, 120]
Generated capacity: 60

Fractional Knapsack Value: 240.0
Fractions of items taken: [1, 0.25, 1]

0-1 Knapsack Value: 220
Items selected (0 or 1): [1, 0, 1]
```

In this case, the fractional algorithm results in a higher total value by taking 0.25 of the second item. The 0-1 solution, although optimal under its constraints, produces a slightly lower total value due to its inability to divide items.

Task 3 Problem Greedy + Dynamic (coin change):

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

Here's a simple greedy algorithm approach:

1. Sort the coins in descending order.
2. Iterate through each coin and take as many as possible of the highest denomination without exceeding the total.
3. Subtract the value of the used coins from the total.
4. Repeat until the total becomes zero.

```python
def greedy_coin_change(coins, total):
    coins.sort(reverse=True)  # Sort coins in descending order
    coin_count = {}

    for coin in coins:
        if total >= coin:
            count = total // coin
            total -= count * coin
            coin_count[coin] = count

    return coin_count
```

For example, we have the following input data:

```python
# Example usage:
coins = [1, 5, 10, 20]
total = 47
result = greedy_coin_change(coins, total)
print("Coin usage:", result)
```

Following the proposed algorithm, it should first take two coins of 20, then 1 of 5 and finally 2 of 1 in order to return the total of 47, minimizing the coins used:

```
Coin usage: {20: 2, 5: 1, 1: 2}
```

2-3. Propose an optimal global solution for the minimum number of coins required

In some coin systems, the greedy approach doesn't always yield the optimal solution because it doesn't consider future combinations that could minimize the total number of coins.

In the coin change problem, dynamic programming (DP) finds the minimum number of coins needed to form a target amount `N`. To achieve this, two lists are used: `dp` and `last_used_coin`. The list `dp[i]` keeps track of the minimum number of coins needed to reach the amount `i`, while `last_used_coin[i]` stores the value of the last coin used to reach that amount.

The algorithm starts by initializing `dp[0] = 0`, since no coins are needed to make zero, while all other entries are set to infinity (`inf`). For each amount from `1` to `N`, the algorithm checks every available coin. If using a particular coin results in fewer coins needed, it updates `dp[i]` and records the coin in `last_used_coin[i]`.

For example, with coins `[1, 5, 11]` and a target total `N = 15`, the algorithm builds the table incrementally. It finds that `dp[5] = 1` by using one `5`-coin, and `dp[11] = 1` by using one `11`-coin. For `dp[15]`, the best solution is three `5`-coins (`5 + 5 + 5 = 15`), which results in `dp[15] = 3`.

Once the table is complete, the algorithm reconstructs the exact coins used by backtracking from the total `N`. Starting from `dp[15]`, it checks which coin was last used (in this case, a `5`-coin) and subtracts its value from the total. This process continues until the total is reduced to zero. The result is the list `[5, 5, 5]`, representing the coins used to reach `15`.

In the end, for the coin system `[1, 5, 11]` and a total of `15`, the algorithm returns that the minimum number of coins needed is `3`, and the coins used are `[5, 5, 5]`. This method guarantees both the **optimal number of coins**, even in coin systems where a greedy algorithm would fail.

```python
def dynamic_coin_change(coins, total):
    dp = [float('inf')] * (total + 1)
    dp[0] = 0  # No coins needed for 0

    # Track which coin was last used to reach each amount
    last_used_coin = [-1] * (total + 1)

    for i in range(1, total + 1):
        for coin in coins:
            if coin <= i and dp[i - coin] + 1 < dp[i]:
                dp[i] = dp[i - coin] + 1
                last_used_coin[i] = coin

    # Reconstruct the coins used
    if dp[total] == float('inf'):
        return -1, []  # No solution found

    coins_used = []
    current = total
    while current > 0:
        coin = last_used_coin[current]
        coins_used.append(coin)
        current -= coin

    return dp[total], coins_used
```

```python
# Example usage:
coins = [1, 5, 11]
total = 15
min_coins, coins_used = dynamic_coin_change(coins, total)
print(f"Minimum number of coins needed: {min_coins}")
print(f"Coins used: {coins_used}")
```

```
Minimum number of coins needed: 3
Coins used: [5, 5, 5]
```

4. Is the Norwegian Coin System Greedy?

The Norwegian coin system uses coins with denominations [1, 5, 10, 20]. A coin system is considered greedy if the greedy algorithm always provides the optimal solution (i.e., the fewest coins needed) for any total amount.

To determine whether the Norwegian coin system is greedy, we can test a few amounts using the greedy algorithm and compare the results with those from the dynamic programming (DP) approach. If both algorithms consistently return the same result for all possible amounts, the system is greedy.

Example Tests:

- Total = 23
    - Greedy Solution: {20: 1, 1: 3}, 4 coins
    - Dynamic Programming Solution: {20: 1, 1: 3}, 4 coins
- Total = 40
    - Greedy Solution: {20: 2}, 2 coins
    - Dynamic Programming Solution: {20: 2}, 2 coins
- Total = 17
    - Greedy Solution: {10: 1, 5: 1, 1: 2}, 4 coins
    - Dynamic Programming Solution: {10: 1, 5: 1, 1: 2}, 4 coins

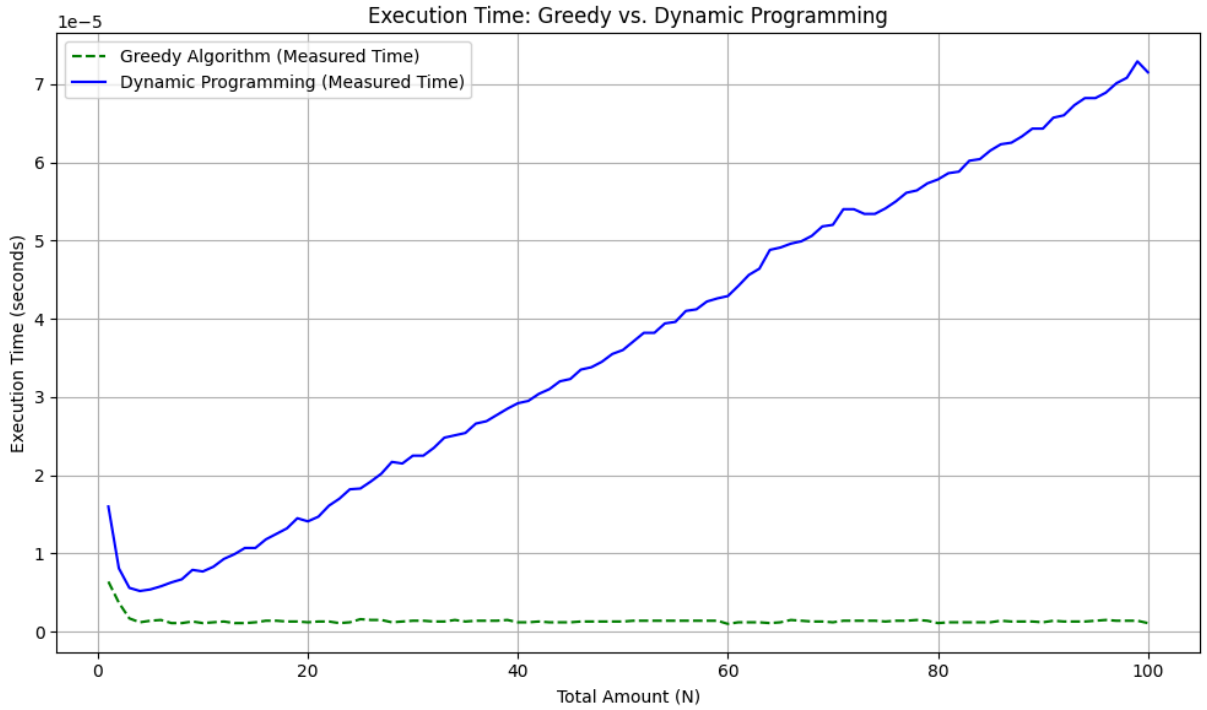
In all cases, both the **greedy algorithm** and the **dynamic programming solution** yield the same minimum number of coins. Since no counterexample exists where the greedy algorithm fails to produce the optimal result, the Norwegian coin system can be considered **greedy**.

This means that, for the Norwegian currency system [1, 5, 10, 20], the greedy algorithm always provides the minimum number of coins required to make any total amount.

5. Running Time of the Two Algorithms

The running time of both the greedy algorithm and the dynamic programming (DP) algorithm differs significantly based on their approaches.

To find out the execution time of each one, a new python file executes both algorithms on the Norwegian currency coin system and varying the total (N) from 1 to 100. The execution times are stored and finally displayed on a graph:



- Greedy Algorithm Time Complexity:

The greedy algorithm iterates through each coin denomination, starting from the largest and subtracting it from the total until the remaining amount is zero. The coins are sorted once (if necessary), and then each denomination is checked:

- **Time Complexity:**
  - **O(k log k)** if sorting the coins is needed, where `k` is the number of coin denominations.
  - **O(k)** if the coins are already sorted, as it simply iterates through the list once.

Actual Execution Time:

As shown in the graph, the greedy algorithm's execution time remains almost constant for different total amounts `N`. This is because the algorithm's performance depends on the number of coin denominations (`k`), not the total amount. In systems like the Norwegian coin system `[1, 5, 10, 20]`, the greedy algorithm is efficient and provides both fast and optimal solutions.

- Dynamic Programming Algorithm Time Complexity:

The dynamic programming (DP) approach builds a table of size `N+1`, where each entry represents the minimum number of coins needed to form that total:

- **Time Complexity:**
  - **O(N * k)** since for each amount from `1` to `N`, it checks all `k` denominations.

Actual Execution Time:

The graph shows that the execution time increases linearly as `N` increases, which aligns with the theoretical time complexity. While slower than the greedy algorithm for large totals, it guarantees the **optimal solution** for any coin system, even when the greedy approach fails.

- Conclusion: Which Algorithm is Faster?

On the one hand, the **greedy algorithm** is significantly faster and more efficient, with execution time independent of the total amount `N`. It's the best choice for systems like Norway's, where the greedy approach always gives an optimal solution.

On the other hand, the **dynamic programming algorithm**, though slower, guarantees an optimal solution for any coin system. It becomes essential when dealing with non-greedy systems where the greedy algorithm could fail.

GitHub Files

https://github.com/AntonMG4/DAT600_AlgorithmTheory/tree/main/Assignment2