# Algorithm Analysis Report

Harry CHICHEPORTICHE , Theo DE MORAIS

March 9, 2025

# 1 Matrix Chain Multiplication

## 1.1 Solve the Parenthesization Problem by Hand

Consider the matrices:

- $A_1 : 10 \times 30$
- $A_2 : 30 \times 5$
- $A_3 : 5 \times 60$
- $A_4 : 60 \times 15$
- $A_5 : 15 \times 10$

Using the recursive formula:

$$m[i,j] = \min_{i \leq k < j} \{m[i,k] + m[k+1,j] + p_{i-1}p_k p_j\} \tag{1}$$

We construct the memoization table step by step:

| $i \backslash j$ | 1 | 2 | 3 | 4 | 5 |
|---|---|---|---|---|---|
| 1 | 0 | 1500 | 4500 | 10500 | 12000 |
| 2 | | 0 | 9000 | 13500 | 16500 |
| 3 | | | 0 | 4500 | 6000 |
| 4 | | | | 0 | 2250 |
| 5 | | | | | 0 |

The optimal split points stored in $s$ are:

| $i \backslash j$ | 2 | 3 | 4 | 5 |
|---|---|---|---|---|
| 1 | 1 | 2 | 2 | 4 |
| 2 | | 2 | 3 | 3 |
| 3 | | | 3 | 4 |
| 4 | | | | 4 |

Thus, the optimal parenthesization is: **((A1 (A2 A3)) (A4 A5))**.

### 1.2 Dynamic Programming Implementation

```python
import sys

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

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

def main():
    p = [10, 30, 5, 60, 15, 10]
    m, s = matrix_chain_order(p)
\if __name__ == "__main__":
    main()
```

### 1.3 Greedy Approach

There is no greedy choice that applies because the problem exhibits optimal substructure but not the greedy-choice property.

## 2 Fractional and 0-1 Knapsack

### 2.1 0-1 Knapsack Problem Implementation

The 0-1 Knapsack Problem is solved using dynamic programming. The goal is to compute the maximum value that can be obtained without exceeding the knapsack's capacity. The following Python code implements the 0-1 knapsack problem:

```python
def knapsack_01(weights, values, capacity):
    n = len(weights)
    dp = [[0] * (capacity + 1) for _ in range(n + 1)]

    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], values[i - 1] + dp[i - 1][
    w - weights[i - 1]])
            else:
                dp[i][w] = dp[i - 1][w]
    return dp[n][capacity]
```

The dynamic programming approach fills the table based on whether or not to include an item in the knapsack. The result is stored in a table, and the maximum value is returned at the bottom-right corner.

## 2.2 Fractional Knapsack Problem Implementation

The Fractional Knapsack Problem is solved using a greedy approach. The items are sorted based on their value-to-weight ratio, and we select the items with the highest ratio to maximize the value of the knapsack.

```
def fractional_knapsack(weights, values, capacity):
    items = sorted(zip(weights, values), key=lambda x: x[1] / x[0],
   reverse=True)
    max_value = 0
    for weight, value in items:
        if capacity >= weight:
            max_value += value
            capacity -= weight
        else:
            max_value += (value / weight) * capacity
            break
    return max_value
```

The greedy algorithm ensures that we maximize the total value by selecting items in the order of their value-to-weight ratio, considering fractions of items when necessary.

## 2.3 Conclusion

Both the 0-1 and Fractional Knapsack Problems are essential in optimization tasks. The 0-1 knapsack problem requires dynamic programming for an exact solution, while the fractional knapsack problem can be solved more efficiently using a greedy approach. These implementations provide solutions to two variations of the knapsack problem, each suited to different kinds of constraints.

# 3 Greedy + Dynamic (Coin Change Problem)

### Problem Statement

Given an array of coin denominations $c_1 < c_2 < \cdots < c_n$, the objective is to determine the **fewest coins** needed to achieve a total sum $N$.

### 1. Greedy Solution

The greedy algorithm selects the largest coin denomination that does not exceed the remaining amount $N$. It repeats this process until the remaining amount becomes zero.

#### Algorithm:

- Sort the coins in decreasing order.
- At each step, pick the largest coin less than or equal to the remaining amount.
- Subtract its value from the remaining amount.
- Repeat until the amount is zero.

**Example:** Coin denominations: $\{1, 5, 10, 25\}$
Target sum: $N = 63$

- Pick 25: remaining 38
- Pick 25: remaining 13
- Pick 10: remaining 3
- Pick 1, 1, 1: remaining 0

**Total coins used: 6.**

```python
def greedy_coin_change(coins, N):
    coins.sort(reverse=True)
    result = []
    for coin in coins:
        while N >= coin:
            N -= coin
            result.append(coin)
    return result
```

## 2. Limitation of Greedy Solution

The greedy algorithm does not always yield an optimal solution. Consider the following example:

**Coin denominations:** $\{1, 5, 11\}$

**Target sum:** $N = 15$

**Greedy choice:**

- Pick 11: remaining 4
- Pick 1, 1, 1, 1: total of 5 coins.

**Optimal solution:**

- Pick 5, 5, 5: total of 3 coins.

The greedy algorithm fails because it makes local optimal choices without considering global optimality.

## 3. Dynamic Programming Solution

Dynamic programming guarantees an optimal solution for any currency system. It builds a solution by solving all subproblems from 1 to $N$.

**Idea:**

- Create an array $dp$ where $dp[i]$ represents the minimum number of coins needed to make amount $i$.
- Initialize $dp[0] = 0$.
- For each $i$ from 1 to $N$, compute:

$$dp[i] = \min_{c_j \leq i} (dp[i - c_j] + 1)$$

```python
def dp_coin_change(coins, N):
    dp = [float('inf')] * (N + 1)
    dp[0] = 0

    for i in range(1, N + 1):
        for coin in coins:
            if i - coin >= 0:
                dp[i] = min(dp[i], dp[i - coin] + 1)

    return dp[N] if dp[N] != float('inf') else -1
```

**Example:**

- Coin denominations: $\{1, 5, 11\}$
- Target sum: $N = 15$
- $dp[15] = 3$ (three 5 coins)

## 4. Is the Norwegian Coin System Greedy-Optimal?

**Norwegian coins:** $\{1, 5, 10, 20\}$

**Test cases:**

- $N = 30$: Greedy picks $20 + 10 = 30$ (2 coins) $\Rightarrow$ optimal.
- $N = 23$: Greedy picks $20 + 1 + 1 + 1 = 23$ (4 coins) $\Rightarrow$ optimal.
- $N = 40$: Greedy picks $20 + 20 = 40$ (2 coins) $\Rightarrow$ optimal.

**Conclusion:** The Norwegian coin system is **greedy-optimal**, because:

- Every larger coin is either a multiple of a smaller one, or no better combination exists.
- The greedy algorithm always yields the optimal result for any amount.

## 5. Running Time Analysis

- **Greedy Algorithm:**
  - Time complexity: $O(N/c_{\max})$ (in worst case, $N$ divided by largest coin).
  - Space complexity: $O(1)$
  - Limitation: Not guaranteed to be optimal for all currency systems.
- **Dynamic Programming Algorithm:**
  - Time complexity: $O(N \times m)$, where $m$ is the number of coin denominations.
  - Space complexity: $O(N)$
  - Advantage: Always returns the optimal solution.

## Conclusion

- The greedy algorithm is simple and fast but only works with specific coin systems.
- Dynamic programming guarantees an optimal solution and works for all coin systems but requires more computation time and memory.
- The Norwegian coin system allows the greedy algorithm to always produce optimal solutions.