Dynamic Programming

-used to solve optimization problems (problems where we have to find min or max solution)

-applicable to problems that can be divided into optimal subproblems

-the sequence of computation must be organized in such a way that later computed solutions can use earlier computed solutions

Approach:

-break down the complex problem into simpler subproblems

-find the optimal solution to these subproblems

-store the result of the subproblems (memoization)

-reuse the results so that the subproblem is not recomputed

-finally calculates the result of the complex problem (returns an optimal value plus, if needed, an optimal solution)

->subproblems are not independent and overlapping

-problem of recursive fibonacci program: we compute the subfunctions more than one time (->not optimal, exponential time complexity can be decreased to linear time with dp) -> overlapping of subproblems can be fixed with storage of data

->time complexity is high because we recompute the same subproblem multiple times, but instead we could store the value of the subproblem and use it for bigger problems (much better in terms of time performance, not good for space complexity since we have to store lots of data)

-we can store the values of the subproblems in an array and use it for bigger problems. Memorization technique is known as top-down approach (be start from big subproblems to small/trivial subproblems)

The bottom-up approach trades space for time

-another memorization technique is the tabulation method

Matrix-Chain Multiplication

-Given a sequence of matrices, find in what order they should be multiplied such that the cost of the multiplication is minimum

-We have many options to multiply a chain of matrices because matrix multiplication is associative

– Multiplying p × q matrix A and q × r matrix B takes p · q · r multiplications; result is a

p × r matrix.

-the order in which we parenthesize the product affects the cost of the multiplication:

For example, suppose A is a 10 × 30 matrix, B is a 30 × 5 matrix, and C is a 5 × 60 matrix. Then,

(AB)C = (10×30×5) + (10×5×60) = 1500 + 3000 = 4500 operations ->minimum cost

A(BC) = (30×5×60) + (10×30×60) = 9000 + 18000 = 27000 operations.

If 4 matrices A, B, C, D we can find final result in 5 ways A(B(CD)) or A((BC)(D)) or (AB)(CD) or ((AB)C)D or (A(BC))D.

General formula to find number of ways we can find solution is **(2n)! / [(n+1)! n!]** (n is the number of multiplications, not the number of matrices!)

Recursive Algorithm (has exponential time which is too slow):

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

Dynamic Programming approach:

Store the optimal cost M(i,j) for each subproblem in a two-dimensional array M[1…n, 1…n]. Initially all entries will be set to ∞.

```
MATRIX-CHAIN(i, j)
    IF T[i][j] < ∞ THEN return T[i][j]
    IF i = j THEN T[i][j] = 0, return 0
    m = ∞
    FOR k = i to j − 1 DO
        q = MATRIX-CHAIN(i, k) + MATRIX-CHAIN(k + 1, j) + p_{i−1} · p_k · p_j
        IF q < m THEN m = q
    OD
    T[i][j] = m
    return m
END MATRIX-CHAIN

return MATRIX-CHAIN(1, n)
```

Time Complexity

If there are n number of matrices we are creating a table contains [(n) (n+1)] / 2 cells that is in worst case total number of cells n*n = n^2 cells we need calculate = **O (n^2)**

For each one of entry we need find minimum number of multiplications taking worst (it happens at last cell in table) that is Table [1,4] which equals to **O (n)** time.

Finally **O (n^2) * O (n) = O (n^3)** is the time complexity.

Space Complexity

We are creating a table of n x n so space complexity is **O (n^2)**

General Notes:

-cost of multiplying a matrix with itself is 0 (M[i,i] = 0, length = 0) -> diagonal of the table

-length = 1. We multiply a Matrix with another distinct matrix -> there is only one option of multiplication

-length = 2. We multiply a Matrix with the two next matrices -> there are multiple options (depends on where we put the parenthesis)

-length = 3. We multiply a Matrix from Ai..4 -> there are multiple options (depends on where we put the parenthesis)

Longest Common Subsequence (LCS)

-from 2 given strings, gives the longest sequence of characters that the two strings have in common (sequence doesn't have to be contiguous)

If S1 and S2 are the two given sequences then, Z is the common subsequence of S1 and S2 if Z is a subsequence of both S1 and S2. Furthermore, Z must be a strictly increasing sequence of the indices of both S1 and S2.

In a strictly increasing sequence, the indices of the elements chosen from the original sequences must be in ascending order in Z

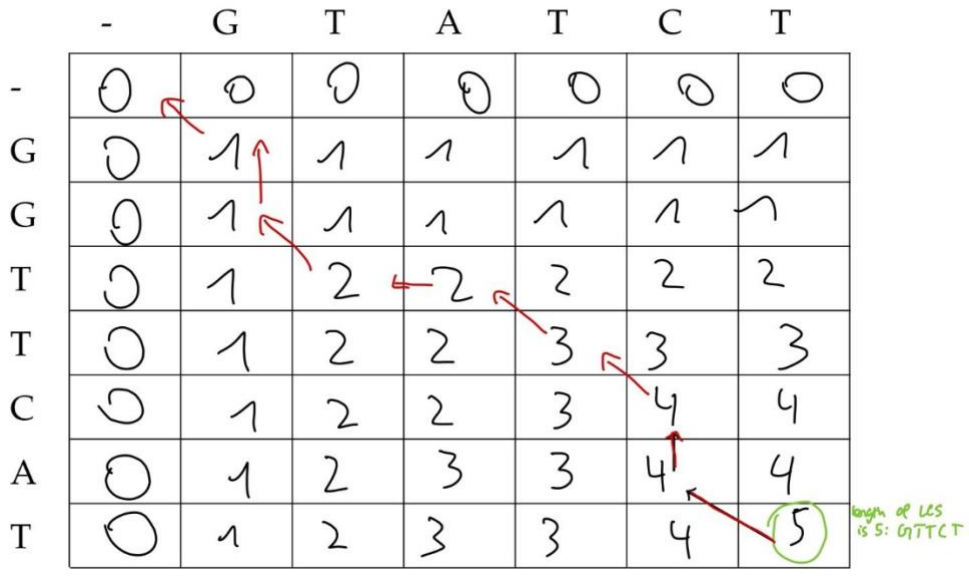
If the last character of the strings matches:

-LCS[i][j] = LCS[i-1][j-1] + 1

If the last character doesn't match:

-LCS[i][j] = max(LCS[i-1][j], LCS[i][j-1])

Determine the length of LCS ("GGTTCAT", "GTATCT").



Coin Changes

-assume that you have coins of different denominations (e.g. 5, 15, 20, 50)

-assume that of each denomination you have an unlimited number of coins

-Given an amount A and m denominations the goal is to change A with as few coins as possible

$$c(i) = \begin{cases} 0 & \text{if } i = 0 \\ \min_{1 \leq k \leq m} \{1 + c(i - d[k])\} & \text{if } i > 0 \end{cases}$$