More dynamic programming

Dynamic programming solutions rely on the optimal substructure property. Usually the recursive solutions to these problems take exponential time with many redundant calculations because the subproblems are not independent.

One more dynamic programming example from Chapter 15 that we will cover:

- Matrix-Chain Product

Matrix-Chain Product

If A is an m×n matrix and B is an n×p matrix, then

\[ A \cdot B = C \]

is an m×p matrix and the time needed to compute C is \(O(mnp)\).

- there are \(mp\) elements of C
- each element of C requires \(n\) scalar multiplications and \(n-1\) scalar additions

Matrix-Chain Multiplication Problem:

Given matrices \(A_1, A_2, A_3, \ldots, A_n\), where the dimension of \(A_i\) is \(p_{i-1} \times p_i\), determine the minimum number of multiplications needed to compute \(A_1 \cdot A_2 \cdot \ldots \cdot A_n\). This involves finding the optimal way to parenthesize the matrices.

For more than 2 matrices, there exists more than one order of multiplication.

Matrix-Chain Product Example

\[ A_1 (4 \times 2) \]
\[ A_2 (2 \times 5) \]
\[ A_3 (5 \times 1) \]

The order of mult can make a difference in # of steps

Two ways to parenthesize this product:

- \((A_1 (A_2 \cdot A_3))\)
- \((A_1 \cdot (A_2 \cdot A_3))\)

Matrix-Chain Product – Recursive Solution

The optimal substructure of this problem can be given with the following argument:

Suppose an optimal way to parenthesize \(A_{i_1} \ldots A_{i_t}\) splits the product between \(A_{i_k}\) and \(A_{i_{k+1}}\). Then the way the prefix subchain \(A_{i_1} \ldots A_{i_k}\) is parenthesized must be optimal. Why?

If there were a less costly way to parenthesize \(A_{i_1} \ldots A_{i_t}\) substituting that solution as the way to parenthesize \(A_{i_1} \ldots A_{i_t}\) gives a solution with lower cost, contradicting the assumption that the way the original group of matrices was parenthesized was optimal.

Therefore, the structure of the subproblem solutions must be optimal.
Longest Common Subsequence Problem (§ 15.4)

**Problem:** Given $X = < x_1, x_2, ..., x_m >$ and $Y = < y_1, y_2, ..., y_n >$, find a longest common subsequence (LCS) of $X$ and $Y$.

**Example:**

$X = (A, B, C, B, D, A, B)$

$Y = (B, D, C, A, B, A)$

$LCS_X = (B, C, B, A)$ or $LCS_Y = (B, D, A, B)$

**Brute-Force solution:**

1. Enumerate all subsequences of $X$ and check to see if they appear in the correct order in $Y$ (chars in sequences are not necessarily consecutive).
2. Each subsequence of $X$ corresponds to a subset of the indices $(1, 2, ..., m)$ of the elements of $X$, so there are $2^m$ subsequences of $X$ to be enumerated.
3. Clearly, this is not a good approach—time to try dynamic programming!

Recursive Solution to LCS Problem

1. **Let $C[i,j]$ be the length of the LCS of $X_{i..j}$ and $Y_{i..j}$**.
2. **Our goal:** $C[1,m] = C[1,n]$
3. **Basic:** $C[i,j] = 0$ and $C[i,0] = 0$
4. $C[i,j]$ is calculated as shown below (two cases):

**Case 1:** $x_i = y_j (i,j > 0)$

In this case, we can increase the size of the LCS of $X_{i..j}$ and $Y_{i..j}$ by one by appending $x_i$ to the LCS of $X_{i..j}$ and $Y_{i..j}$, i.e.,

$C[i,j] = C[i-1,j-1] + 1$

**Case 2:** $x_i \neq y_j (i,j > 0)$

In this case, we take the LCS to be the longer of the LCS of $X_{i..j}$ and $Y_{i..j}$, i.e.,

$C[i,j] = \max(C[i-1,j], C[i,j-1])$

Complexity:

- $O(n^2)$ time because of the nested loops with each of $i, j,$ and $k$ taking on at most $n-1$ values.
- $O(n^2)$ space for two $n \times n$ matrices $M$ and $s$.
Bottom-Up LCS DP
Running time = O(mn) (constant time for each entry in C)
This algorithm finds the value of the LCS, but how can we keep track of the characters in the LCS?
We need to keep track of which neighboring table entry gave the optimal solution to a sub-problem (break ties arbitrarily).
if $x_i = y_j$ the answer came from the upper left (diagonal),
if $x_i \neq y_j$ the answer came from above or to the left, otherwise, whichever value is larger (if equal, default to above).

Complexity of LCS Algorithm
The running time of the LCS algorithm is $O(mn)$, since each table entry takes $O(1)$ time to compute.
The running time of the Print-LCS algorithm is $O(m + n)$, since one of $m$ or $n$ is decremented in each stage of the recursion.

Bottom-Up DP Solution to LCS Problem
To compute $C[i, j]$, we need the solutions to:
- $C[i-1, j-1]$ (when $x_i = y_j$)
- $C[i-1, j]$ and $C[i, j-1]$ (when $x_i \neq y_j$)

We need an $m$ by $n$ matrix to store results.

Bottom-Up LCS DP
Running time = O(mn) (constant time for each entry in C)
This algorithm finds the value of the LCS, but how can we keep track of the characters in the LCS?
We need to keep track of which neighboring table entry gave the optimal solution to a sub-problem (break ties arbitrarily).
if $x_i = y_j$ the answer came from the upper left (diagonal),
if $x_i \neq y_j$ the answer came from above or to the left, otherwise, whichever value is larger (if equal, default to above).

Construction of the LCS
Initial call is Print-LCS(B,X,len(X),len(Y)), where B is the arrow table.