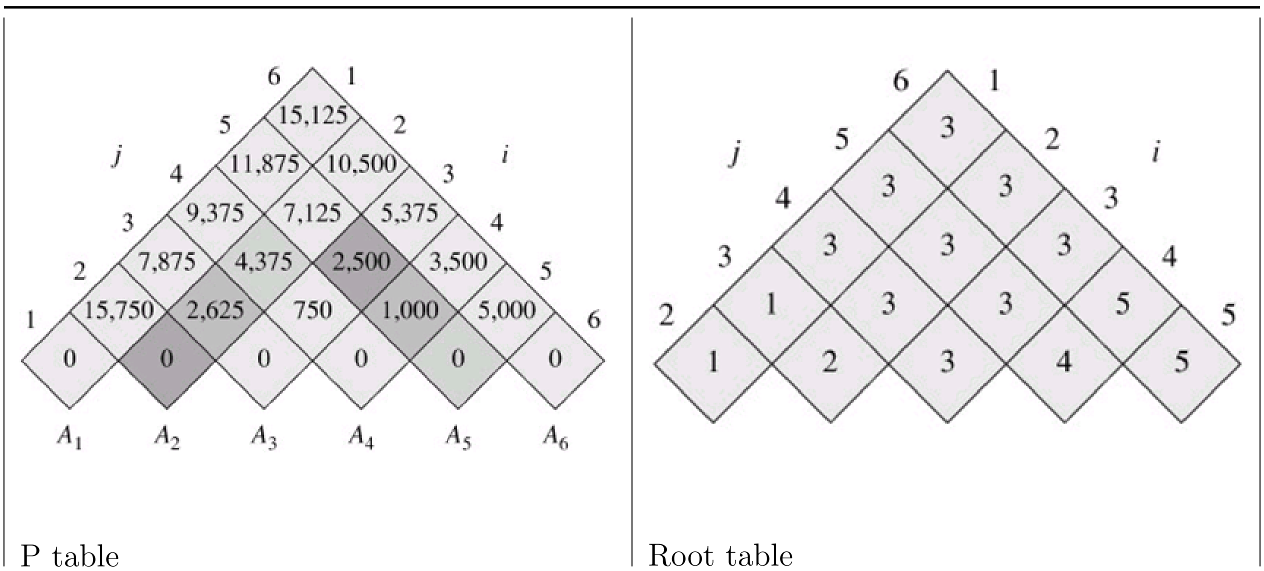
Opening Questions

1. Why are optimal solutions to sub-problems stored in dynamic programming solutions? You only have to store the optimal answer for the subproblem because optimal substructure says that the overlapping subproblems only needs to know the optimal solution, not any other solution.

Constructing the answer for the Optimal Matrix Chain Multiplication (optimal parenthesization) from the dynamic programming table. See the solution to the example problem

$$\begin{array}{cccccc} A_1 & A_2 & A_3 & A_4 & A_5 & A_6 \\ 30 \times 35 & 35 \times 15 & 15 \times 5 & 5 \times 10 & 10 \times 20 & 20 \times 25 \end{array}$$



2. Write the optimal parenthesization.

$$\begin{array}{ccc} A_1 A_2 & A_3 A_4 & A_5 A_6 \\ (A_1 A_2 & A_3)(A_4 & A_5 A_6) \\ ((A_1)(A_2 & A_3))(A_4 & A_5 A_6) \\ ((A_1)(A_2 & A_3))((A_4 & A_5) A_6) \end{array}$$

3. Write pseudocode to use the root table to print the optimal parenthesization. Then write pseudocode to use the root table to actually perform the multiplications in the optimal parenthesization.

Algorithm 15.1 Print Optimal Parenthesization for Matrix Chain Multiplication

```
function PrintOptParens(root, from, to)    ▷ Initial Call: PrintOptParens(root, 1, n)
    if from == to then
        Print("A_from")
    else
        Print("(")
        PrintOptParens(root, from, root[from,to])    ▷ The root[from, to] states the index that was extracted from the root table.
        Print("×")
        PrintOptParens(root, root[from,to]+1, to)
        Print(")")
    end if
end function
```

Assume we have a function MultMatrix(x, y) that multiplies 2 matrices and returns the results(given correct dimensions.)

Algorithm 15.2 Multiply Optimal Parenthesization for Matrix Chain Multiplication

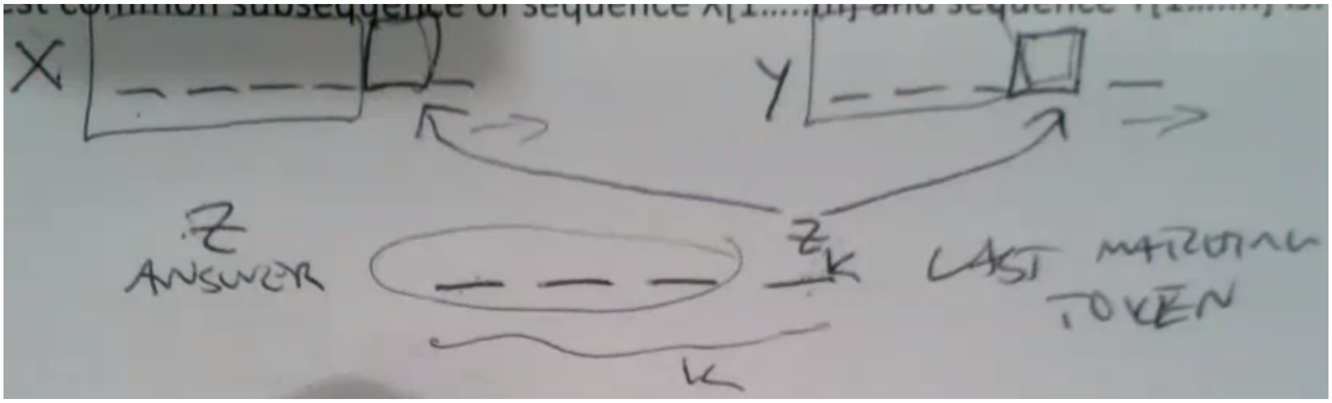
```
function MultOptParens(root, from, to)    ▷ Initial Call: MultOptParens(root, 1, n)
    if from == to then
        return A_from
    else
        a ← MultOptParens(root, from, root[from,to])
        b ← MultOptParens(root, root[from,to] + 1, to)
        return MultMatrix(a, b)
    end if
end function
```

Longest Common Subsequence

Example: $X[1 \ldots m]$ $\quad Y[1 \ldots n]$
$X = ABCBDAB$ $\quad Y = BDCABA$
Length of LCS = 4 $BCBA$ or $BCAB$

1. The brute force (try all possibilities) approach would be to find all subsequences of one input, see if each exists in other input. How many are there? Assume $m < n$. You would have the choices: $\frac{2}{1}\ \frac{2}{2}\ \frac{2}{3}\ \frac{2}{4} \ldots \frac{2}{m-2}\ \frac{2}{m-1}\ \frac{2}{m}$, where the two represents that there are 2 choices for each position, yes or no, including all no's representing the empty set. This means there 2^m possible subsequences, again assuming that m is smaller than n. For each generated subsequence, you have to check if it's in Y, which would take $O(n)$ for each check. The total runtime for brute force is $O(n2^m)$.

2. Step 1: Generically define the structure of the optimal solution to the Longest Common Subsequence problem. The longest common sequence of sequence $X[1 \ldots m]$ and sequence $Y[1 \ldots n]$ is:



Algorithm 15.3 Basic pseudocode for Longest Common Sequence

function LONGESTCOMMONSEQUENCE(x, m, y, n)
    if $x[m] == y[n]$ **then** ▷ Use it
        Add 1 to the answer for the subproblem $x[1 \ldots (m-1)]$,$y[1 \ldots (n-1)]$
    else ▷
        $\text{Max}\begin{cases} \text{Answer for } x[1 \ldots m], y[1 \ldots (n-1)] \\ \text{Answer for } x[1 \ldots (m-1)], y[1 \ldots n] \end{cases}$
    end if
end function

3. Step 2: Recursively define the optimal solution. Assume $C(i, j)$ is the optimal answer for up to position i in X and position j in Y. Make sure you include the base case.

$$C(i,j) = \begin{cases} C(i-1, j-1) & \text{if } x[i] == y[j] \\ \max\begin{cases} C(i-1, j) \\ C(i, j-1) \end{cases} & \text{if } x[i] \neq y[j] \end{cases}$$

Base Cases:

- $i == 0$
  - $C(0, j) = 0$
  - $C(i, 0) = 0$
  - $C(0, 0) = 0$

4. Use proof by contradiction to show that Longest Common Subsequence problem has optimal substructure, i.e. the optimal answer to problem must contain optimal answers to subproblems. Assume $z[1 \ldots k]$ is optimal for $x[1 \ldots i]$ and $y[1 \ldots j]$. If $z[k] = x[i] = y[j]$, then we have a subproblem $z[1 \ldots (k-1)]$ that must be optimal for $x[1 \ldots (i-1)]$ $y[1 \ldots (j-1)]$. The contradiction is that if someone said some sequence w with length k (or more) is optimal for $x[1 \ldots (i-1)]$, $y[1 \ldots (j-1)]$, which is longer than z, then you could add on the end of w the characters that we know matches ($x[i] = y[j]$), then we would get a solution larger than k for this problem. But that's impossible because we assumed $z[1 \ldots k]$ is the longest possible solution.

5. Step 3: Compute solution using a table bottom up for the Longest Common Subsequence problem. Use your answer to question 3 above. Note the overlapping sub-problems as you go.

					Y			
C			B	D	C	A	B	A
		0	0	0	0	0	0	0
	A	0	→↓ 0	→↓ 0	→↓ 0	↘1	→1	↘ 1
	B	0	↘ 1	→ 1	→ 1	→↓ 1	↘ 2	→ 2
	C	0	↓ 1	→↓ 1	↘ 2	→ 2	→↓ 2	→↓ 2
X	B	0	↘ 1	→↓ 1	↓ 2	→↓ 2	↘ 3	→ 3
	D	0	↓ 1	↘ 2	→↓ 2	→↓ 2	↓ 3	→↓ 3
	A	0	↓ 1	↓ 2	→↓ 2	↘ 3	→↓ 3	↘ 4
	B	0	↘ 1	↓ 2	→↓ 2	↓ 3	↘ 4	→↓ 4

6. Step 4: Construct Optimal Solution Walking backwards and adding the character when there was a match, there are 3 answers. $BCBA$, $BDAB$, $BCAB$ all of length 4.

$X = ABCBDAB \qquad Y = BDCABA$

`https://www.cs.usfca.edu/~galles/visualization/DPLCS.html`