Dynamic Programming

**Step 1:** Characterize the structure of an optimal solution. An optimal solution to an instance of the problem contains optimal solutions to subproblem instances.

**Step 2:** Recursively define the value of an optimal solution in terms of the optimal solutions to the subproblems.

**Step 3:** Compute the value of an optimal solution, typically in a bottom-up fashion.

**Step 4:** Construct an optimal solution from the computed information.

Example: Fully Parenthesizing Matrix-chain Multiplication

**Step 1:** Given an optimal parenthesization \((A_1A_2\ldots A_k)(A_{k+1}A_{k+2}\ldots A_n)\), the parenthesization of the subchain \(A_1A_2\ldots A_k\) and the parenthesization of \(A_{k+1}A_{k+2}\ldots A_n\) should be optimal.

**Step 2:** \(m_{ij}\): minimum number of scalar multiply-add operations needed to compute \(A_iA_{i+1}\ldots A_j\)

\[
m_{ij} = \begin{cases} 
0 & \text{if } i = j \\
\min_{i<k<j} \{m_{ik} + m_{k+1,j} + p_i-1p_kp_j\} & \text{if } i < j
\end{cases}
\]

**Step 3:**

```
MATRIX-CHAIN-ORDER(\(p\))
    \(n = p.length - 1\)
    let \(m[1..n,1..n]\) and \(s[1..n-1,2..n]\) be new tables
    for \(i ← 1\) to \(n\)
        \(m[i,i] ← 0\)
    for \(l ← 2\) to \(n\)
        for \(i ← 1\) to \(n - l + 1\)
            \(j ← i + l - 1\)
            \(m[i,j] ← ∞\)
            for \(k ← i\) to \(j - 1\)
                \(q ← m[i,k] + m[k+1,j] + p_i-1p_kp_j\)
                if \(q < m[i,j]\)
                    \(m[i,j] ← q\)
                    \(s[i,j] ← k\)
        return \(m\) and \(s\)
```

**Step 4:**

```
PRINT-OPTIMAL-PARENS(\(s, i, j\))
    if \(i = j\)
        print “A”
i
    else
        print “(“
        PRINT-OPTIMAL-PARENS(\(s, i, s[i, j]\))
        PRINT-OPTIMAL-PARENS(\(s, s[i, j] + 1, j\))
        print “)”
```

Greedy Algorithms:

**Greedy Choice Property**
- A greedy choice at each step yields a globally optimal solution
- The proof examines a globally optimal solution:
  - It shows that the solution can be modified so that a greedy choice made as the first step reduces the problem to a similar but smaller problem
  - Then induction is applied to show that a greedy choice can be used at each step

**Optimal Substructure**
- A problem exhibits optimal substructure if an optimal solution to the problem contains optimal solutions to subproblems within it
**STRONGLY-CONNECTED-COMPONENTS(G)**

1. call DFS(G) to compute finishing times \( f[u] \) for each vertex \( u \)
2. construct \( G^2 \) and then call DFS\((G^2)\), but in the main loop of DFS, process the vertices in order of decreasing \( f[u] \) computed in step (1)
3. output the vertices of each tree in the depth-first forest formed in line 3 as a separate SCC