**Floyd-Warshall APSP Algorithm**

**Floyd-Warshall-APSP(A)**
1. \( n = \text{rows}[A] \)
2. \( D^{(0)} = A \)
3. for \( k = 1 \) to \( n \) do
4. \hspace{1em} for \( i = 1 \) to \( n \) do
5. \hspace{2em} for \( j = 1 \) to \( n \) do
6. \hspace{3em} \( d_{ij}^{(k)} = \text{min} \left( d_{ij}^{(k-1)}, d_{ik}^{(k-1)} + d_{kj}^{(k-1)} \right) \)
7. return \( D^{(n)} \)

**Complexity of FW-APSP**

Running time:
- Lines 3 – 6: \( |V|^3 \) time for triply-nested for loops
- Overall = \( \Theta(V^3) \)

Additionally, the code is tight, with no elaborate data structures and so the constant hidden in the \( \Theta \)-notation is small.

Space usage is \( O(V^3) \), since we create \( V \) matrices of size \( O(V^2) \) each.
However the space usage can be reduced to \( O(V^2) \) because we only need to keep 1 matrix for the entire algorithm.

**Transitive Closure**

(modified Floyd-Warshall APSP)

The transitive closure of \( G \) is the graph \( G^* = (V, E^*) \), where \( E^* = \{(i, j) : \text{there is a path from vertex } i \text{ to vertex } j \text{ in } G\} \)

One way to solve the transitive closure problem is to assign edge weights of 1 to each edge in \( G \) and run the Floyd-Warshall algorithm. If there is a path from \( i \) to \( j \) in \( G \), we get \( d_{ij} < n \), otherwise, we get \( d_{ij} = \infty \).

**Transitive Closure Algorithm**

Use adjacency matrix \( T \) for \( G = (V, E) \):
\[
T[i, j] = t_{ij} = \begin{cases} 1 & \text{if } (i, j) \in E \text{ or if } i = j \\ 0 & \text{if } i = j \text{ and } (i, j) \notin E \end{cases}
\]

After the execution, the zeroes in the matrix indicate, for each \( d_{ij} \), that \( j \) is unreachable from \( i \).
Matrix-Chain Product

If A is an \( m \times n \) matrix and B is \( n \times p \), then \( A \cdot B = C \) is \( m \times p \) and the time needed to compute C is \( O(mnp) \).

- There are \( mp \) elements of C.
- Each one 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 \( d_i \times d_i \), determine the minimum number of multiplications needed to compute the product \( A_1 \cdot A_2 \cdot A_3 \cdots \cdot A_n \). This involves finding the optimal parenthesization of the matrices.

**Matrix-Chain Product – Recursive Solution**

\[ RMP(d, i, j) = \begin{cases} 
0, & \text{if } i = j \\
\min_{i < k < j} \{ RMP(d, i, k) + RMP(d, k+1, j) + d_i \cdot d_k \cdot d_j \}, & \text{otherwise}
\end{cases} \]

1. If \( i = j \), then return 0.
2. \( M[i,j] = \infty \).
3. For \( k = i \) to \( j-1 \) do.
4. \( q = RMP(d, i, k) + RMP(d, k+1, j) + d_i \cdot d_k \cdot d_j \).
5. If \( q < M[i,j] \), then \( M[i,j] = q \).
6. Return \( M[i,j] \).

**Example**

\( \begin{pmatrix} A_1 & (4 \times 2) & A_2 & (2 \times 5) \end{pmatrix} \)

\( d = (4, 2, 5, 1) \)

The order of multiplication can make a difference.

Two ways to parenthesize this product:

- \((A_1 \cdot A_2) \cdot A_3\) total multiplications = 40 + 20 = 60
- \((A_1 \cdot A_2) \cdot A_3\) requires 2 \cdot 5 \cdot 1 = 10 multiplications, \( M_1 \) is 2 \times 1 matrix.

**Matrix-Chain Product – Recursive Solution**

\[ M[i,j] = \min_{i < k < j} \{ M[i,k] + M[k+1,j] + d_i \cdot d_k \cdot d_j \} \]

1. If \( i = j \), then return 0.
2. \( M[i,j] = \infty \).
3. For \( k = i \) to \( j-1 \) do.
4. \( q = M[i,k] + M[k+1,j] + d_i \cdot d_k \cdot d_j \).
5. If \( q < M[i,j] \), then \( M[i,j] = q \).
6. Return \( M[i,j] \).

**Example**

\( \begin{pmatrix} A_1 & (4 \times 2) & A_2 & (2 \times 5) & A_3 & (5 \times 1) \end{pmatrix} \)

\( d = (4, 2, 5, 1) \)

Let \( M[i,j] = \min \{ M[i,k] + M[k+1,j] + d_i \cdot d_k \cdot d_j \} \), where dimension of \( A_1, A_2, \ldots, A_n \) is \( d_1, d_1, \ldots, d_n \), \( M[i,j] = 0 \) for \( i = 1 \) to \( n \), and \( M[1,n] \) is the solution we want.

**Dynamic Programming (Ch. 15)**

Development of a dynamic-programming algorithm can be broken into 4 steps:

1. Characterize the structure of an optimal solution
2. Recursively define the value of an optimal solution
3. Compute the value of an optimal solution from the bottom up
4. Construct an optimal solution from computed information

Dynamic programming can provide a good solution for problems that take exponential time to solve by brute-force methods.

For example, a brute-force approach to solving the APSP problem is to enumerate and test the length of all possible paths between \( |V| \) nodes. The running time of this approach is \( O(2^{|V|}) \). Floyd-Warshall does much better.
Matrix-Chain Product – Recursive Solution

The exact solution to this recurrence is \( \Omega \left( \frac{4^n}{n^{3/2}} \right) \), which is exponential in \( n \), the number of matrices.

Let's see if dynamic programming can do better.

Matrix-Chain Product – Recursive Top-Down Solution

The difference between this and the recursive solution we saw earlier is that this algorithm looks up the solution to subproblems in lower rows of the table when solving higher rows.

Matrix-Chain Product – Top-Down Solution

Complexity:
- \( O(n^3) \) space for \( M \)
- \( O(n^2) \) time since each recursive call generates \( O(n) \) others.

This algorithm has no recursive calls.

What makes a problem a DP candidate?

1. Simple subproblems: Must be able to break the global optimization problem into subproblems, each having a similar structure to the original problem.

2. Optimal substructure: An optimal solution contains within it the optimal solution to subproblems. We should not be able to find a global solution that contains sub-optimal subproblems.

For example, the optimal solution that splits the matrix product between \( A_k \) and \( A_{k+1} \) contains within it optimal solutions to subproblems of parenthesizing \( A_1 \cdots A_k \) and \( A_{k+1} \cdots A_n \) above diagonal only.

Complexity:
- \( O(n^2) \) space for \( M \)
- \( O(n^2) \) time since each recursive call generates \( O(n) \) others.

In general, if all subproblems must be solved at least once, a bottom-up approach usually out-performs a top-down memo-ized algorithm by a constant factor due to less overhead for recursion.

Memo-ization versus bottom-up recursion
What makes a problem a DP candidate?

3. **Overlapping subproblems**: A recursive solution to the problem solves the same sub-problems multiple times.