Matrix Chain Products

- Matrix chain multiplication
  - Associative: \((AB)C = A(BC)\)
  - Compute \(A = A_0A_1 \ldots A_{n-1}\)
  - Size of \(A_i\): \(d_i \times d_{i+1}\)
  - Problem: We want to find a way to compute the result with the minimal number of operations.

- Example
  - Size of \(B\): \(3 \times 100\)
  - Size of \(C\): \(100 \times 5\)
  - Size of \(D\): \(5 \times 5\)
  - \((BC)D\): \((3 \times 100 \times 5) + (3 \times 5 \times 5) = 1500 + 75 = 1575\) ops
  - \(B(CE)\): \((3 \times 100 \times 5) + (100 \times 5 \times 5) = 1500 + 2500 = 4000\) ops

Matrix Chain Products

- Matrix Multiplication
  - \(C = AB\)
  - Size of \(A\): \(d \times e\)
  - Size of \(B\): \(e \times f\)
  - \(AB\) takes \(d \times e \times f\) times of basic operations.

  \[
  C[i,j] = \sum_{k=1}^{e} A[i,k]B[k,j]
  \]

Brute-Force Algorithm

- An enumeration approach
- Matrix chain product algorithm
  - Try all possible ways to parenthesize \(A = A_0A_1 \ldots A_{n-1}\).
  - Calculate number of operations for each one.
  - Pick the one that is best.

- Running time
  - The number of parenthesizations is equal to the number of binary trees with \(n\) nodes.
  - This is an exponential number.
  - This is a terrible algorithm!
Greedy Algorithm

- Choose the local optimal iteratively
- Repeatedly select the product that uses the fewest operations.
- Example: computation of $ABCD$
  - Size of $A$: 10 × 5
  - Size of $B$: 5 × 10
  - Size of $C$: 10 × 5
  - Size of $D$: 5 × 10
  - $AB$ (10 × 5 × 10) vs. $BC$ (5 × 10 × 5) vs. $CD$ (10 × 5 × 10)
  - $A((BC)D)$: 500 + 250 + 250 = 1000 ops

Dynamic Programming

- Concept
  - Primarily for optimization problems that may require a lot of time otherwise
  - Simplifying a complicated problem by breaking it down into simpler subproblems in a recursive manner
- Two key observations:
  - The problem can be split into subproblems.
  - The optimal solution can be defined in terms of optimal subproblems.
- Condition for dynamic programming
  - Simple subproblems: the subproblems can be defined in terms of a few variables.
  - Subproblem optimality: the global optimum value can be defined in terms of optimal subproblems
  - Subproblem overlap: the subproblems are not independent, but instead they overlap (hence, should be constructed bottom-up).

Another example for $ABCD$

- Size of $A$: 101 × 11
- Size of $B$: 11 × 9
- Size of $C$: 9 × 100
- Size of $D$: 100 × 99
- The greedy approach
  - Best solution: $A((BC)D)$
  - Number of operations: $(101 \times 11 \times 99) + (11 \times 9 \times 100) + (11 \times 100 \times 99) = 109989 + 9900 + 108900 = 228789$
- Another method
  - A candidate solution: $(AB)(CD)$
  - Number of operations: $(101 \times 11 \times 9) + (9 \times 100 \times 99) + (101 \times 9 \times 99) = 9999 + 89100 + 89991 = 189090$

The greedy approach does not give us an optimal solution necessarily.

Dynamic Programming for Matrix Chain Product

- Problem definition
  - Whole problem: computation of $A = A_0A_1 \ldots A_{n-1}$
  - Find the best parenthesization of $A_0A_{i+1} \ldots A_j$.
  - Let $N_{i,j}$ denote the number of operations done by this subproblem.
  - The optimal solution for the whole problem is $N_{0,n-1}$.
- Approach
  - There has to be a final multiplication (root of the expression tree) for the optimal solution; for example, $(A_0 \ldots A_i)(A_{i+1}A_{n-1})$
  - The optimal solution $N_{0,n-1}$ is the sum of two optimal subproblems, $N_{0,i}$ and $N_{i+1,n-1}$ plus the time for the last multiplication.
  - We perform this procedure in a recursive manner.
A Characterizing Equation

• Let us consider all possible places for that final multiply:
  ▪ Recall that $A_i$ is a $d_i \times d_{i+1}$ dimensional matrix.
  ▪ So, a characterizing equation for $N_{i,j}$ is the following:
    
    $N_{i,j} = \min_{i \leq k < j} \{N_{i,k} + N_{k+1,j} + d_id_{k+1}d_{j+1}\}$
    
    Note that subproblems overlap and hence we cannot divide the problem into completely independent subproblems (divide and conquer).

• Bottom-up computation
  ▪ Base case: $N_{i,j} = 0$
  ▪ $N_{i,j+1} = N_{i,j} + d_id_{j+1}d_{j+2}$
  ▪ $N_{i,j+2} = \min \left\{ \begin{array}{l}
N_{i,j} + N_{i+1,j+2} + d_id_{j+1}d_{j+2}, \\
N_{i,j+1} + N_{i+2,j+2} + d_id_{j+2}d_{j+2}
\end{array} \right\}$
  ▪ $N_{i,j+3} \ldots$
  ▪ Until you get all $N_{i,j}$

Visualization of Dynamic Programming

• Table construction
  ▪ The bottom-up construction fills the upper diagonal matrix.
  ▪ $N_{i,j}$ gets values from previous entries in $i$-th row and $j$-th column
  ▪ Filling in each entry in the table takes $O(n)$ time.
  ▪ Total running time: $O(n^3)$
  ▪ Getting actual parenthesization can be done by remembering $k$ for each entry.

A Dynamic Programming Algorithm

• Characteristics
  ▪ Since subproblems overlap, we don’t use recursion.
  ▪ Instead, we construct optimal subproblems bottom-up.
  ▪ $N_{i,i}$’s are easy, so start with them.
  ▪ Then do subproblems with larger lengths.
  ▪ The running time is $O(n^3)$.

<table>
<thead>
<tr>
<th>Algorithm matrixChain(S):</th>
</tr>
</thead>
<tbody>
<tr>
<td>Input: sequence $S$ of $n$ matrices to be multiplied</td>
</tr>
<tr>
<td>Output: number of operations in an optimal paranethization of $S$</td>
</tr>
<tr>
<td>for $i \leftarrow 0$ to $n-1$ do</td>
</tr>
<tr>
<td>$N_{i,i} \leftarrow 0$</td>
</tr>
<tr>
<td>for $b \leftarrow 1$ to $n-1$ do</td>
</tr>
<tr>
<td>for $i \leftarrow 0$ to $n-b-1$ do</td>
</tr>
<tr>
<td>$j \leftarrow i+b$</td>
</tr>
<tr>
<td>$N_{i,j} \leftarrow \infty$</td>
</tr>
<tr>
<td>for $k \leftarrow i$ to $j-1$ do</td>
</tr>
<tr>
<td>$N_{i,j} \leftarrow \min{N_{i,j}, N_{i,k} + N_{k+1,j} + d_id_{k+1}d_{j+1}}$</td>
</tr>
</tbody>
</table>

Similarity between Strings

• A common text processing problem:
  ▪ Two strands of DNA
  ▪ Two versions of source code for the same program
  ▪ diff: a built-in program for comparing text files in unix or linux.
The Longest Common Subsequence

• Subsequences
  - A string of the form \( x_{i_1}x_{i_2} \ldots x_{i_k} \) from a character string \( x_0x_1 \ldots x_{n-1} \), where \( i_j < i_{j+1} \).
  - Not necessary contiguous but taken in order
  - Not the same as substring!

• Examples
  - Original string: ABCDEFGHIJK
  - Subsequence: ACEGIJK, DFGHK
  - Not subsequence: DAGH

• Longest Common Subsequence (LCS) problem
  - Given two strings \( X \) and \( Y \), the longest common subsequence (LCS) problem is to find a longest subsequence common to both \( X \) and \( Y \).

A Poor Approach to the LCS Problem

• A Brute-force solution:
  - Enumerate all subsequences of \( X \)
  - Test which ones are also subsequences of \( Y \)
  - Pick the longest one.

• Analysis:
  - If \( X \) is of length \( n \), then it has \( 2^n \) subsequences.
  - If \( Y \) is of length \( m \), the time complexity is \( O(2^n m) \)
  - This is an exponential time algorithm!

A Dynamic Programming Approach

• Procedure
  - Define \( L[i,j] \) to be the length of the longest common subsequence of \( X [0 \ldots i] \) and \( Y [0 \ldots j] \).
  - Allow for -1 as an index, so \( L[-1,k] = 0 \) and \( L[k,-1] = 0 \) to indicate that the null part of \( X \) or \( Y \) has no match with the other.
  - Then we can define \( L[i,j] \) in the general case as follows:
    - If \( x_i = y_j \), then \( L[i,j] = L[i-1,j-1] + 1 \) \quad Match
    - If \( x_i \neq y_j \), then \( L[i,j] = \max\{L[i-1,j], L[i,j-1]\} \) \quad No match

Visualizing the LCS Algorithm
Analysis of LCS Algorithm

• We have two nested loops
  ▪ The outer one iterates $n$ times.
  ▪ The inner one iterates $m$ times.
  ▪ A constant amount of work is done inside each iteration of the inner loop.
  ▪ Thus, the total running time is $O(nm)$.

• Solution
  ▪ The final answer is contained in $L[n, m]$.
  ▪ The subsequence can be recovered from the table by backtracking.

Pseudocode of LCS Algorithm

```
Algorithm LCS(X, Y):

Input: Strings X and Y with n and m elements, respectively
Output: For $i = 0, ..., n-1$, $j = 0, ..., m-1$, the length $L[i, j]$ of a longest string
that is a subsequence of both the string $X[0..i] = x_0x_1x_2...x_i$ and
the string $Y[0..j] = y_0y_1y_2...y_j$

for $i = 0$ to $n-1$ do
  $L[i, -1] = 0$
for $j = 0$ to $m-1$ do
  $L[-1, j] = 0$
for $i = 0$ to $n-1$ do
  for $j = 0$ to $m-1$ do
    if $x_i = y_j$ then
      $L[i, j] = L[i-1, j-1] + 1$
    else
      $L[i, j] = \max(L[i-1, j], L[i, j-1])$

return array $L$
```