Dynamic Programming

Example 1: Longest Common Subsequence (LCS)

- Given two sequences x[1..m] and y[1..n], find a longest subsequence common to them both.

```
x: A B C B D A B
y: B D C A B A
```

Different phrasing: Find a set of a maximum number of segments, such that
- Each segment connects a character of x to an identical character of y,
- Each character is used at most once
- Segments do not intersect.

Brute-force LCS algorithm

Check every subsequence of x[1..m] to see if it is also a subsequence of y[1..n].

Analysis
- Checking = $\Theta(m+n)$ time per subsequence.
- $2^m$ subsequences of x (each bit-vector of length m determines a distinct subsequence of x).
- Worst-case running time = $\Theta((m+n)2^m)$ = exponential time.

Towards a better algorithm

Simplification:
1. Look at the length of a longest common subsequence.
2. Extend the algorithm to find the LCS itself.

Strategy: Consider prefixes of x and y.

Notation: Denote the length of a sequence s by |s|.

Recursive formulation

**Theorem.**

\[
c[i,j] = \begin{cases} 
  c[i-1,j-1] + 1 & \text{if } x[i] = y[j], \\
  \max\{c[i-1,j], c[i,j-1]\} & \text{otherwise.}
\end{cases}
\]

Proof: First Note that it is impossible that
x[i] is matched to an element in y[1..j-1] and in addition
y[j] is matched to an element in x[1..i-1].

Recursive formulation-cont

Case (I): x[i] = y[j].

```
|   | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |
\hline
x: | 1 | 2 | 3 | 4 |   |   |   |   |   |   |
\hline
y: | 1 | 2 |   |   |   |   |   |   |   |   |
```

We claim that there is a max matching that matches x[i] to y[j].

Indeed, if x[i] is matched to y[j] (for k=j) then y[j] is unmatched (otherwise we have two crossing segments). Hence we can obtain another matching of the same cardinality by match x[i] to y[j].

This implies that we can match x[1..i-1] to y[1..j-1], and add the match (x[i],y[j]). So $c[i,j] = c[i-1,j-1] + 1$. 

Recursive formulation-cont
Recursive formulation- cont

Case (II): \( x[i] \neq y[j] \)

Recall \( \text{LCS}(x[1 \ldots i], y[1 \ldots j]) \) it cannot be that both \( x[i] \) and \( y[j] \) are both matched.

Claim: \( c[i,j] = \max \{ c[i-1,j], c[i,j-1] \} \)

If \( x[i] \) is unmatched then \( \text{LCS}(x[1 \ldots i], y[1 \ldots j]) = \text{LCS}(x[1 \ldots i-1], y[1 \ldots j]) \)

If \( y[j] \) is unmatched then \( \text{LCS}(x[1 \ldots i], y[1 \ldots j]) = \text{LCS}(x[1 \ldots i], y[1 \ldots j-1]) \)

So \( c[i,j] = \max \{ c[i-1,j], c[i,j-1] \} \)

Recursive algorithm for LCS

\[
\text{LCS}(x, y, i, j) \\
\text{if } (i==0 \text{ or } j==0) \text{ return } 0 \\
\text{if } x[i] = y[j] \text{ then return } \text{LCS}(x, y, i-1, j-1) + 1 \\
\text{else return } \max \{ \text{LCS}(x, y, i-1, j), \text{LCS}(x, y, i, j-1) \}  \\
\]

To call the function \( \text{LCS}(x, y, m, n) \)

**Worst-case:** \( x[i] \neq y[j] \), for all \( i,j \) in which case the algorithm evaluates two subproblems, each with only one parameter decremented.

Dynamic-programming hallmark #1

**Optimal substructure**

An optimal solution to a problem (instance) contains optimal solutions to subproblems.

If \( z = \text{LCS}(x, y) \), then any prefix of \( z \) is an LCS of a prefix of \( x \) and a prefix of \( y \).

Dynamic-programming hallmark #2

**Overlapping subproblems**

A recursive solution contains a “small” number of distinct subproblems repeated many times.

The number of distinct LCS subproblems for two strings of lengths \( m \) and \( n \) is only \( mn \).

Memoization algorithm

**Memoization:** After computing a solution to a subproblem, store it in a table. Subsequent calls check the table to avoid redoing work.

\[
\text{LCS}(x, y) \\
\text{for } i=0 \text{ to } m \quad c[i, 0] = 0 \\
\text{for } j=0 \text{ to } n \quad c[0, j] = 0 \\
\text{for } i=1 \text{ to } m \\
\text{for } j=1 \text{ to } n \\
\text{if } (x[i] = y[j]) \text{ then } c[i,j] \leftarrow c[i-1,j-1] + 1 \\
\text{else } c[i,j] \leftarrow \max \{ c[i-1,j], c[i,j-1] \} \\
\]

Time \( = \Theta(mn) \) = constant work per table entry. Space \( = \Theta(mn) \).
Example 1.

\[ X = \text{BCAB} \]
\[ Y = \text{DCBDA} \]

\[ \text{LCS(X,Y)} = \text{BCBA} \]

\[ \begin{array}{cccccccc}
   & 1 & 2 & 3 & 4 & 5 & 6 & 7 \\
1 & 0 & |0 & 0 & 0 & 0 & 0 & 0 \\
2 & B & 0 & 1 & 1 & 1 & 1 & 1 \\
3 & C & 0 & 1 & 1 & 1 & 2 & 2 \\
4 & D & 0 & 1 & 2 & 2 & 3 & 3 \\
5 & A & 0 & 1 & 2 & 3 & 3 & 4 \\
6 & A & 0 & 1 & 2 & 3 & 3 & 4 \\
\end{array} \]

Reconstruction \( z = \text{LCS}(x,y) \)

- IDEA: Compute the table bottom-up. Fill \( z \) backward.

- Observation: \( c[i][j] \geq \min(c[i-1][j], c[i][j-1]) \)

- Proof Sketch: We use a longer prefix, so there are more chars to be matched.

\[ \begin{array}{cccccccc}
   & 1 & 2 & 3 & 4 & 5 & 6 & 7 \\
1 & 0 & |0 & 0 & 0 & 0 & 0 & 0 \\
2 & B & 0 & 1 & 1 & 1 & 1 & 1 \\
3 & D & 0 & 1 & 1 & 1 & 2 & 2 \\
4 & C & 0 & 1 & 2 & 2 & 2 & 2 \\
5 & A & 0 & 1 & 2 & 2 & 3 & 3 \\
6 & B & 0 & 1 & 2 & 3 & 3 & 4 \\
7 & A & 0 & 1 & 2 & 3 & 3 & 4 \\
\end{array} \]

Matrix Chain-Products

- Review: Matrix Multiplication.
- \( A = AB \)
- \( A \) is \( d \times e \), \( B \) is \( e \times f \)

- \( O(\text{def}) \) time

\[ C[i,j] = \sum_{k=0}^{e} A[i,k] \times B[k,j] \]

Example 2 of dynamic programming:

- Pick the one that is best

An Enumeration Approach

- Matrix Chain-Product Alg.:
  - Try all possible ways to parenthesize
  - Calculate number of ops for each one
  - Pick the one that is best

- Running time:
  - \# of parenthesizations = \# of binary trees with \( n \) nodes
  - Exponential!
  - Called the \( n^\text{th} \) Catalan number – it is almost \( 4^n \).
  - This is a terrible algorithm!
A Greedy Approach

Repeatedly select the product that uses the fewest operations.

Counter-example:
- A is 101 × 11
- B is 11 × 9
- C is 9 × 100
- D is 100 × 99
  - Idea selects A(BC)(D) 109989+9900+108900=228789 mults
  - Best is (AB)(CD) 9999+89991+89100=189090 mults

A “Recursive” Approach

- Define subproblems:
  - Find the best parenthesization of $A_i A_{i+1} \ldots A_j$.
  - Let $N_{i,j}$ be # of operations done by this subproblem.
  - The optimal solution for the whole problem is $N_{0,n}$.

- Subproblem optimality: Assume the last multiplication taken place is multiplying $(A_0 \ldots A_k) (A_{k+1} \ldots A_n)$.
  - Then the optimal solution $N_{i,j}$ is the sum of two optimal subproblems, $N_{i,k} + N_{k+1,j}$ plus the time for the last multiply.
  - If the global optimum did not have these optimal subproblems, we could define an even better “optimal” solution.

A Characterizing Equation

- Again assume the last multiplication is $A_i A_{i+1} \ldots A_j$.
  - That is, we break at index $i$.
- Consider all possible places for that final multiply (possible values of $0 \leq i \leq n-1$ ). That is…
  - $(A_0 (A_1 A_2 \ldots A_i), (A_0 A_{i+1} (A_{i+2} \ldots A_n))$, and
  - $(A_0 A_{i+1}) (A_{i+2} \ldots A_n)$, $(A_0 A_{i+1} (A_{i+2} \ldots A_n)$ etc till
  - $(A_0 A_{i+1}) (A_{i+2} \ldots A_n)$.
- 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:

$$ N_{i,j} = \min_{i < k < j} \{N_{i,k} + N_{k+1,j} + d_i d_k d_{j+1}\} $$

I.e., break $(A_1 \ldots A_j)$ into $(A_1 \ldots A_i) (A_{i+1} \ldots A_j)$.

A Dynamic Programming Algorithm

Since subproblems overlap, we don’t use recursion. Instead, we construct optimal subproblems “bottom-up.”

- $N_{i,j}$ are easy, so start with them.
- Then do length 2,3,… subproblems, and so on.

Running time: $O(n^3)$

A Dynamic Programming Algorithm Visualization

The bottom-up construction fills in the $N$ array by diagonals.

- $N_{i,j}$ gets values from previous entries in $i$-th row and $j$-th column.
- Filling in each entry in the $N$ table takes $O(n)$ time.
- Total run time: $O(n^3)$
- Getting actual parenthesization can be done by remembering “LC” for each $N$ entry (next slide).

Matrix Chain algorithm

How do we find the actual order of operations?

Example: ABCD

<table>
<thead>
<tr>
<th>N</th>
<th>A</th>
<th>0</th>
<th>0</th>
<th>0</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>B</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>C</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>3</td>
<td>D</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

Output: Parenthesization is $(A(B(CD)))$
The General Dynamic Programming Technique

- Applies to a problem that at first seems to require a lot of time (often exponential), provided we have:
  - **Simple subproblems**: the subproblems can be defined in terms of a few variables, such as \( j, k, l, m \), and so on.
  - **Subproblem optimality**: the global optimum value can be defined in terms of optimal subproblems.

### Floyd-Warshall’s Algorithm

**Algorithm AllPairs\( D_{ij} \)** for all vertex pairs \((i,j)\)

- if \( i = j \) then \( D_{ii} \leftarrow 0 \)
- else if \((v_i, v_j)\) is an edge in \( G \)
  - \( D_{ij} \leftarrow w(v_i, v_j) \)
- else \( D_{ij} \leftarrow \infty \)

for \( k \leftarrow 1 \) to \( n \) do

for \( i \leftarrow 1 \) to \( n \) do

for \( j \leftarrow 1 \) to \( n \) do

\[
D_{ij} = \min\{ D_{ik} + D_{kj} \}
\]

return \( D_{ij} \)
Example 4: Edit distance

Given strings \(x, y\), the edit distance \(ed(x, y)\) between \(x\) and \(y\) is defined as the minimum number of operations that we need to perform on \(x\), in order to obtain \(y\).

**Definition:** An operation (in this context) is an insertion/deletion/replacement of a single character.

Examples:

- \(ed(\text{"aaba"}, \text{"aaba")} = 0\)
- \(ed(\text{"aaa"}, \text{"aab"}) = 1\)
- \(ed(\text{"aaa"}, \text{"abaa")} = 1\)
- \(ed(\text{"baaa"}, \text{"")} = 4\)
- \(ed(\text{"baaa"}, \text{"aab"}) = 2\)

Example 4': ``Priced'' Edit distance \(ed(x, y)\)

Assume also given:

- \(\text{InsCost}\) - the cost of a single insertion into \(x\).
- \(\text{DelCost}\) - the cost of a single deletion from \(x\), and
- \(\text{RepCost}\) - the cost of replacing one character of \(x\)

by a different character.

**Definition:** Given strings \(x, y\), the edit distance \(ed(x, y)\) between \(x\) and \(y\) is the cheapest sequence of operations, starting on \(x\) and ending at \(y\).

**Problem:** Compute \(ed(x, y)\), and compute the sequence of operations.

**Theorem:**

Let \(c[i,j] = ed(x[1..i], y[1..j])\) then

If \(x[i] = y[j]\) then \(c[i,j] = c[i-1,j-1]\)

If \(x[i]\neq y[j]\) then \(c[i,j] = \min( c[i,j-1] + \text{InsCost}, c[i-1,j] + \text{DelCost}, c[i-1,j-1] + \text{RepCost})\)

**Algorithm**

**Memoization:** After computing a solution to a subproblem, store it in a table. Subsequent calls check the table to avoid redoing work.

```plaintext
for i=0 to m
  c[i,0] = i * DelCost
for j=0 to n
  c[0,j] = j * InsCost
for i=1 to m
  for j=1 to n
    if (x[i] = y[j])
      then \(c[i,j] = c[i-1,j-1]\)
    else \(c[i,j] = \min( c[i,j-1] + \text{InsCost}, c[i-1,j] + \text{DelCost}, c[i-1,j-1] + \text{RepCost})\)
```

Time = \(O(mn)\) = constant work per table entry. Space = \(O(mn)\).