**Dynamic Programming**

Slides courtesy of Charles Leiserson with small changes by Carola Wenk

---

**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" not "the"

$x$: A B C B D A B

$y$: B D C A B A

BCBA = LCS$(x, y)$

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.

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

**Strategy:** Consider prefixes of $x$ and $y$.
- Define $c[i, j] = |LCS(x[1..i], y[1..j])|$.
- Then, $c[m, n] = |LCS(x, y)|$.

---

**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 $x[1..i-1]$.

---


**Proof:**

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

Indeed, if $x[i]$ is matched to $y[k]$ (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 $x[i+1..j]$ to $y[i+1..j]$ and add the match $(x[i], y[j])$. So $c[i, j] = c[i-1, j-1] + 1$.
Recursive formulation-cont

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

Claim: \( \forall i, j \geq 0 \), \( c[i, j] = \max\{ c[i-1, j], c[i, j-1] \} \)

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

Optimal substructure

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

Dynamic-programming hallmark #1

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 \( mn \).

Recursive algorithm for LCS

\[
\text{LCS}(x, y, i, j) = \\
\begin{cases} 
0 & \text{if } (i=0 \text{ or } j=0) \\
\text{LCS}(x, y, i-1, j-1) + 1 & \text{if } x[i] = y[j] \\
\max\{ \text{LCS}(x, y, i-1, j), \text{LCS}(x, y, i, j-1) \} & \text{otherwise}
\end{cases}
\]

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.

Memoization algorithm

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

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

Time = \( \Theta(mn) \) = constant work per table entry.
Space = \( \Theta(mn) \).
**LCS: Dynamic-programming algorithm**

LCS(X,Y)=“BCBA”

```
<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>B</th>
<th>D</th>
<th>A</th>
<th>B</th>
</tr>
</thead>
<tbody>
<tr>
<td>X=B</td>
<td>D</td>
<td>C</td>
<td>A</td>
<td>B</td>
<td>A</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Y=A</td>
<td>B</td>
<td>C</td>
<td>B</td>
<td>D</td>
<td>A</td>
<td>B</td>
<td></td>
</tr>
</tbody>
</table>
```

**Reconstruction z=LCS(x,y)**

**IDEA:** Compute the table bottom-up. Fill up backward.

**Observation:** \( c[i][j]<c[i+1][j] \) and \( c[i][j]<c[i][j+1] \)

**Proof Sketch:** Use a longer prefix, so there are more chars to be matched.

**LCS Reconstruction:**
Set \( i=m; j=n; k=c[i][j] \)

While(\( i>0 \))

1. if \( c[i][j]<c[i-1][j] \) and \( c[i][j]<c[i][j+1] \) \{ \( \text{if } k=i; j--; k--; \)
2. else \( \text{if } c[i][j]<c[i+1][j] \) or \( c[i][j]<c[i][j+1] \) \{ \( \text{if } k=i+1; j--; \)
3. else \( k--; \)

**Example 2**

of dynamic programming:

**Matrix Chain-Products**

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

- \( O(d^3) \) time

\[
C[i, j] = \sum_{k=0}^{e-1} A[i, k] \cdot B[k, j]
\]

**An Enumeration Approach**

- Matrix Chain-Product Alg.:
  - Try all possible ways to parenthesize \( A_0 A_1 \ldots A_{n+1} \)
  - 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 \( n \)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 \times 11
- B is 11 \times 9
- C is 9 \times 100
- D is 100 \times 99
- Idea selects \( A(BCD) \)
- Best is \( (AB)(CD) \)

A “Recursive” Approach

- Define subproblems:
  - Find the best parenthesization of \( A_i A_{i+1} \ldots A_j \)
  - Let \( N_{ij} \) = # of operations done by this subproblem.
  - The optimal solution for the whole problem is \( N_{0n-1} \).
- Subproblem optimality: Assume the last multiplication taken place is multiplying \( (A_{i-1} \ldots A_j) \).
  - Then the optimal solution \( N_{i-1,j} \) is the sum of two optimal subproblems, \( N_{i-1,k} + N_{k+1,j} \) 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 \( (A_i \ldots A_j) \).
  - 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...
  - \( (A_0 (A_1 A_2 \ldots A_{i-1}) \ldots A_{i+1} \ldots A_j) \)
  - \( (A_0 (A_1 A_2 \ldots A_{i-1}) \ldots A_{i-2} A_{i-1} A_j) \) etc till
  - \( (A_0 A_{i-1} \ldots A_{i-1} A_{i+1} \ldots A_j) \).
- Recall that \( A_i \) is a \( d_i \times d_{i+1} \) dimensional matrix.
  - So, a characterizing equation for \( N_{ij} \) is the following:
    \[
    N_{ij} = \min_{0 \leq k < j} \{ N_{ik} + N_{k+1,j} + d_id_kd_{j+1} \}
    \]
  - i.e, break \( (A_i \ldots A_j) \) into \( (A_i \ldots A_k) (A_{k+1} \ldots A_j) \).

A Dynamic Programming Algorithm

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

\[
N_{ij} \text{'s are easy, so start with them}
\]

Then do length 2,3,... subproblems, and so on.

Running time: \( O(n^3) \)

Algorithm matrixChain(S):

- Input: sequence \( S \) of \( n \) matrices to be multiplied
- Output: \# of multiplications in optimal parenthesization of \( S \)

For \( i \) from 0 to \( n-1 \) do
  \( N_{ii} \leftarrow 0 \)
For \( b \) from 0 to \( n-1 \) do
  \( N_{ij} \leftarrow \infty \)
  //length of a run
  For \( i \) from 0 to \( n-b-1 \) do
    //start of run
    \( j \leftarrow i+b \)
    //end of run
    \( N_{ij} \leftarrow \min( N_{ij} , N_{i,l} + N_{l+1,j} + d_id_ld_{j+1} ) \)
For \( k \) from \( i \) to \( j \) do //break point
  \( N_{ij} \leftarrow \min( N_{ij} , N_{ik} + N_{k+1,j} + d_id_kd_{j+1} ) \)

Matrix Chain algorithm

How do we find the actual order of operations?

Example: ABCD

<table>
<thead>
<tr>
<th>A</th>
<th>0</th>
<th>0</th>
<th>0</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

For \( i \) from 0 to \( n-1 \) do
  \( N_{ii} \leftarrow 0 \)
For \( b \) from 0 to \( n-1 \) do
  \( N_{ij} \leftarrow \infty \)
  //length of a run
  For \( i \) from 0 to \( n-b-1 \) do
    //start of run
    \( j \leftarrow i+b \)
    //end of run
    \( N_{ij} \leftarrow \min( N_{ij} , N_{i,l} + N_{l+1,j} + d_id_ld_{j+1} ) \)
For \( k \) from \( i \) to \( j \) do //break point
  \( N_{ij} \leftarrow \min( N_{ij} , N_{ik} + N_{k+1,j} + d_id_kd_{j+1} ) \)

Return \( N_{0n-1} \)
### Recovering Operations

- **Example:** ABCD
  - A is 10 × 5
  - B is 5 × 10
  - C is 10 × 5
  - D is 5 × 10

<table>
<thead>
<tr>
<th>N</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>50 0</td>
<td>50 0</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>50 0</td>
<td>50 0</td>
<td>50 0</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>50 0</td>
<td>50 0</td>
<td>50 0</td>
<td></td>
</tr>
</tbody>
</table>

- Example 3: All-Pairs Shortest Paths
  - Floyd-Warshall alg
    - Given a graph \( G(V,E) \) with weights (positive and negative) assign to each edge. Assume \( \forall (v_i, v_j) \).
    - Compute a matrix \( D \) such that \( D[i,j] \) contains the length of the shortest path from \( v_i \) to \( v_j \).
    - Define \( P_{i,j}^{(0)} \) as the shortest path \( v_i \rightarrow v_j \) that does not go through any of the vertices \( \{v_i, v_j\} \). (that is, it is allowed to go through any of \( \{v_i, v_j\} \).
    - \( D[i,j] \) – the length of \( P_{i,j}^{(0)} \).
    - We compute \( D_0 \) first, then \( D_1 \) etc.

This example appears in the shortest paths' chapter of CLRS (25.2).

### 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-Pairs Shortest Paths

**Computing \( D[i,j] \) for every \( i,j,k \).**

Algorithm AllPair \( (G) \) for all vertex pairs \( (i,j) \)

if \( i = j \) then \( D[i,j] \leftarrow 0 \)
else if \( (v_i, v_j) \) is an edge in \( G \)
  \( D[i,j] \leftarrow w(v_i, v_j) \)
else \( D[i,j] \leftarrow \infty \)

for \( k \leftarrow 1 \) to \( n \) do
  for \( i \leftarrow 1 \) to \( n \) do
    for \( j \leftarrow 1 \) to \( n \) do
      \( D[i,j] \leftarrow \min\{ D[i,j], D[i,k] + D[k,j] \} \)
      return \( D_0 \)

**Floyd’s algorithm: example**

![Floyd’s algorithm example](image-url)
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 any of the following:
- Insertion/Deletion/Replacement of a single character.

Examples:
- $ed("aaba", "aaba") = 0$
- $ed("aaas", "aba") = 1$
- $ed("baas", "asab") = 2$

**Example 4’:** "Priced" Edit distance $ed(x,y)$

Assume also given:
- $InsCost$, the cost of a single insertion into $x$.
- $DelCost$, the cost of a single deletion from $x$.
- $RepCost$, 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 $x[i]≠y[j]$ then $c(i,j) = \min\{c(i-1,j) + InsCost, c(i,j-1) + DelCost, c(i-1,j-1) + RepCost\}$

**Algorithm**

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

```plaintext
ed(x, y)
for i=0 to m    c[i, 0] = 0
for j=0 to n    c[0, j] = 0
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 if x[i] < y[j]
        then c[i, j] <- c[i-1, j] + InsCost
    else if y[j] < x[i]
        then c[i, j] <- c[i, j-1] + DelCost
    else c[i, j] <- c[i-1, j-1] + RepCost

Time = $Θ(mn)$ = constant work per table entry. Space = $Θ(mn)$.
```