Longest Common Subsequence

• \( C = c_1 \ldots c_g \) is a subsequence of \( A = a_1 \ldots a_m \) if
  \( C \) can be obtained by removing elements from \( A \) (but retaining order)
• \( \text{LCS}(A, B) \): A maximum length sequence that is a subsequence of both \( A \) and \( B \)

\( \text{LCS}('\text{BARTHOLEMES SIMPSON}', 'KRUSTY THE CLOWN') = \text{RTHOWN} \)

LCS Optimization

• \( A = a_1a_2\ldots a_m \)
• \( B = b_1b_2\ldots b_n \)
• \( \text{Opt}[j, k] \) is the length of \( \text{LCS}(a_1a_2\ldots a_j, b_1b_2\ldots b_k) \)

Optimization recurrence

If \( a_j = b_k \), \( \text{Opt}[j, k] = 1 + \text{Opt}[j-1, k-1] \)
If \( a_j \neq b_k \), \( \text{Opt}[j, k] = \max(\text{Opt}[j-1, k], \text{Opt}[j, k-1]) \)

Dynamic Programming Computation

Code to compute \( \text{Opt}[n, m] \)

```c
for (int i = 0; i < n; i++)
    for (int j = 0; j < m; j++)
        if (A[i] == B[j])
            Opt[i, j] = Opt[i-1, j-1] + 1;
        else if (Opt[i-1, j] >= Opt[i, j-1])
            Opt[i, j] := Opt[i-1, j];
        else
            Opt[i, j] := Opt[i, j-1];
```
Storing the path information

\[
\begin{align*}
A[1..m] , B[1..n] \\
\text{for } i \geq 1 \text{ to } m & \quad \text{Opt}[i, 0] := 0; \\
\text{for } j \geq 1 \text{ to } n & \quad \text{Opt}[0, j] := 0; \\
\text{Opt}[0, 0] & := 0; \\
\text{for } i \geq 1 \text{ to } m & \quad \text{for } j \geq 1 \text{ to } n \\
\quad \text{if } A[i] = B[j] & \quad \text{Opt}[i, j] := 1 + \text{Opt}[i-1, j-1]; \quad \text{Best}[i, j] := \text{Diag}; \\
\quad \text{else if } \text{Opt}[i-1, j] \geq \text{Opt}[i, j-1] & \quad \text{Opt}[i, j] := \text{Opt}[i-1, j]; \quad \text{Best}[i, j] := \text{Left}; \\
\quad \text{else} & \quad \text{Opt}[i, j] := \text{Opt}[i, j-1]; \quad \text{Best}[i, j] := \text{Down}; \\
\end{align*}
\]

How good is this algorithm?

- Is it feasible to compute the LCS of two strings of length 300,000 on a standard desktop PC? Why or why not.

Observations about the Algorithm

- The computation can be done in \(O(m+n)\) space if we only need one column of the Opt values or Best Values.
- The algorithm can be run from either end of the strings.

Computing LCS in \(O(nm)\) time and \(O(n+m)\) space

- Divide and conquer algorithm
- Recomputing values used to save space

Divide and Conquer Algorithm

- Where does the best path cross the middle column?
- For a fixed \(i\), and for each \(j\), compute the LCS that has \(a_i\) matched with \(b_j\)

Constrained LCS

- \(\text{LCS}_{i,j}(A,B)\): The LCS such that
  - \(a_1, \ldots, a_i\) paired with elements of \(b_1, \ldots, b_j\)
  - \(a_{i+1}, \ldots, a_m\) paired with elements of \(b_{j+1}, \ldots, b_n\)
- \(\text{LCS}_{4,3}(abbacb, cbbaa)\)
A = RRSSRTTRRTS
B = RTSRRRSTST

Compute LCS\(_{5,0}(A,B)\), LCS\(_{5,1}(A,B)\),…,LCS\(_{5,9}(A,B)\)

<table>
<thead>
<tr>
<th>j</th>
<th>left</th>
<th>right</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>4</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>4</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>3</td>
</tr>
<tr>
<td>3</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>4</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>5</td>
<td>3</td>
<td>2</td>
</tr>
<tr>
<td>6</td>
<td>3</td>
<td>2</td>
</tr>
<tr>
<td>7</td>
<td>3</td>
<td>1</td>
</tr>
<tr>
<td>8</td>
<td>4</td>
<td>1</td>
</tr>
<tr>
<td>9</td>
<td>4</td>
<td>0</td>
</tr>
</tbody>
</table>

Computing the middle column
• From the left, compute LCS\(_{a_1…a_{m/2},b_1…b_j}\)
• From the right, compute LCS\(_{a_{m/2+1…m},b_{j+1…n}}\)
• Add values for corresponding j’s
• Note – this is space efficient

Divide and Conquer
• A = \(a_1…a_m\) \quad B = b_1…b_n
• Find j such that
  – LCS\(_{a_1…a_{m/2},b_1…b_j}\) and
  – LCS\(_{a_{m/2+1…m},b_{j+1…n}}\) yield optimal solution
• Recurse

Algorithm Analysis
• \(T(m,n) = T(m/2, j) + T(m/2, n-j) + cnm\)

Prove by induction that \(T(m,n) <= 2cmn\)
Memory Efficient LCS Summary

- We can afford $O(nm)$ time, but we can’t afford $O(nm)$ space
- If we only want to compute the length of the LCS, we can easily reduce space to $O(n+m)$
- Avoid storing the value by recomputing values
  - Divide and conquer used to reduce problem sizes

Shortest Paths with Dynamic Programming

Shortest Path Problem

- Dijkstra’s Single Source Shortest Paths Algorithm
  - $O(m \log n)$ time, positive cost edges
- General case – handling negative edges
- If there exists a negative cost cycle, the shortest path is not defined
- Bellman-Ford Algorithm
  - $O(mn)$ time for graphs with negative cost edges

Lemma

- If a graph has no negative cost cycles, then the shortest paths are simple paths
- Shortest paths have at most $n-1$ edges

Shortest paths with a fixed number of edges

- Find the shortest path from $v$ to $w$ with exactly $k$ edges

Express as a recurrence

- $\text{Opt}_k(w) = \min_x [\text{Opt}_{k-1}(x) + c_{xw}]$
- $\text{Opt}_0(w) = 0$ if $v=w$ and infinity otherwise
Algorithm, Version 1

```
foreach w:
    M[0, w] = infinity;
    M[0, v] = 0;
    for i = 1 to n-1
        foreach w:
            M[i, w] = min(M[i-1, x] + cost(x, w));
```
Finding negative cost cycles

• What if you want to find negative cost cycles?

Foreign Exchange Arbitrage

<table>
<thead>
<tr>
<th></th>
<th>USD</th>
<th>EUR</th>
<th>CAD</th>
</tr>
</thead>
<tbody>
<tr>
<td>USD</td>
<td>0.8</td>
<td>1.2</td>
<td></td>
</tr>
<tr>
<td>EUR</td>
<td>1.2</td>
<td></td>
<td>1.6</td>
</tr>
<tr>
<td>CAD</td>
<td>0.8</td>
<td>0.6</td>
<td></td>
</tr>
</tbody>
</table>