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)

• $LCS(A, B)$: A maximum length sequence that is a subsequence of both $A$ and $B$

$LCS(\text{BARTHOLEMEWSIMPSON}, \text{KRUSTYTHECLOWN}) = \text{RTHOWN}$
LCS Optimization

- $A = a_1a_2...a_m$
- $B = b_1b_2...b_n$

- $\text{Opt}[j, k]$ is the length of $\text{LCS}(a_1a_2...a_j, b_1b_2...b_k)$
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 Opt[n, m]

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

A[1..m], B[1..n]

for i := 1 to m    Opt[i, 0] := 0;
for j := 1 to n    Opt[0, j] := 0;
Opt[0, 0] := 0;
for i := 1 to m
    for j := 1 to n
        else if Opt[i-1, j] >= Opt[i, j-1]
            {  Opt[i, j] := Opt[i-1, j], Best[i, j] := Left;  }
        else        {  Opt[i, j] := Opt[i, j-1], Best[i, j] := Down;  }
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 \( b_{j+1},\ldots,b_n \)

- \( \text{LCS}_{4,3}(\text{abbacbb}, \text{cbbaa}) \)
\[ A = \textcolor{red}{RRSS}RTT\textcolor{blue}{R}TS \]
\[ B = \textcolor{red}{R}TS\textcolor{blue}{RR}RSSTST \]

Compute \( \text{LCS}_{5,0}(A,B), \text{LCS}_{5,1}(A,B), \ldots, \text{LCS}_{5,9}(A,B) \)
\[
\text{A} = \text{RRSSRTTRTS} \\
\text{B} = \text{RTSRRSTST}
\]

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

<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 $\text{LCS}(a_1 \ldots a_{m/2}, b_1 \ldots b_j)$
- From the right, compute $\text{LCS}(a_{m/2 + 1} \ldots a_m, b_{j+1} \ldots b_n)$
- Add values for corresponding j’s

- Note – this is space efficient
Divide and Conquer

- $A = a_1, \ldots, a_m$ \hspace{0.5cm} $B = b_1, \ldots, b_n$

- Find $j$ such that
  - $LCS(a_1 \ldots a_{m/2}, b_1 \ldots b_j)$ and
  - $LCS(a_{m/2+1} \ldots a_m, b_{j+1} \ldots b_n)$ yield optimal solution

- Recurse
Algorithm Analysis

- $T(m,n) = T(m/2, j) + T(m/2, n-j) + cnm$
Prove by induction that
\[ T(m,n) \leq 2^{cmn} \]
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_x(M[i-1, x] + \text{cost}[x, w]);
Algorithm, Version 2

foreach w
    \( M[0, w] = \text{infinity}; \)
\( M[0, v] = 0; \)
for i = 1 to n-1
    foreach w
        \( M[i, w] = \min(M[i-1, w], \min_x(M[i-1,x] + \text{cost}[x,w])) \)
Algorithm, Version 3

foreach w
    \[ M[w] = \infty; \]
\[ M[v] = 0; \]
for \( i = 1 \) to \( n-1 \)
    foreach w
        \[ M[w] = \min(M[w], \min_x(M[x] + \text{cost}[x,w])); \]
Correctness Proof for Algorithm 3

- Key lemma – at the end of iteration $i$, for all $w$, $M[w] \leq M[i, w]$;

- Reconstructing the path:
  - Set $P[w] = x$, whenever $M[w]$ is updated from vertex $x$
If the pointer graph has a cycle, then the graph has a negative cost cycle

- If \( P[w] = x \) then \( M[w] \geq M[x] + \text{cost}(x,w) \)
  - Equal when \( w \) is updated
  - \( M[x] \) could be reduced after update

- Let \( v_1, v_2, \ldots v_k \) be a cycle in the pointer graph with \((v_k, v_1)\) the last edge added
  - Just before the update
    - \( M[v_j] \geq M[v_{j+1}] + \text{cost}(v_{j+1}, v_j) \) for \( j < k \)
    - \( M[v_k] > M[v_1] + \text{cost}(v_1, v_k) \)
  - Adding everything up
    - \( 0 > \text{cost}(v_1,v_2) + \text{cost}(v_2,v_3) + \ldots + \text{cost}(v_k, v_1) \)
Negative Cycles

• If the pointer graph has a cycle, then the graph has a negative cycle

• Therefore: if the graph has no negative cycles, then the pointer graph has no negative cycles
Finding negative cost cycles

• What if you want to find negative cost cycles?
Foreign Exchange Arbitrage

USD

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