20. Longest Common Subsequence

CPSC 535

Kevin A. Wortman

CALIFORNIA STATE UNIVERSITY FULLERTON

1D vs 2D Dynamic Programming

- Previous class: rod cutting
  - Recursive solution: $r_n = \max_{1 \leq i \leq n}(p_i + r_{n-i})$
  - References a 1D sequence $r_j : r_n = \ldots r_{n-i} \ldots$
  - $\therefore$ initializing $r[j]$ involves one non-nested loop; $\Theta(n)$
  - Call this **1D dynamic programming**
- Now: longest common subsequence (LCS)
  - Recursive solution:
    $c[i][j] = \ldots c[i-1,j-1]+1 \ldots c[i,j-1] \ldots c[i-1,j] \ldots$
  - References a 2D matrix $c[i][j]$
  - $\therefore$ initializing $c[i][j]$ involves two nested loops; $\Theta(n^2)$
  - Call this **2D dynamic programming**

Subsequences

- Let $X = \langle x_1, x_2, \ldots, x_m \rangle$ and $Y = \langle y_1, y_2, \ldots, y_n \rangle$ be two sequences
- Define **prefix** notation: $X_k = \langle x_1, \ldots, x_k \rangle; X_0 = \langle \rangle$
- Informally: a **subsequence** of Y is a copy of Y with some elements removed
- Formally: X is a **subsequence** of Y if there exists an increasing sequence of indices $\langle i_1, i_2, \ldots, i_k \rangle$ such that, for all $j \in [1, k], x_j = y_{i_j}$
- Example: for $X = \langle B, C, D, B \rangle$ and $Y = \langle A, B, C, B, D, A, B \rangle$, X is a subsequence of Y with index sequence $\langle 2, 3, 5, 7 \rangle$

Common Subsequence

- Z is a **common subsequence** of X and Y if Z is a subsequence of both X and Y
- a **longest common subsequence** is a common subsequence of maximum length
- Example: let $X = \langle A, B, C, B, D, A, B \rangle$ (same) and $Y = \langle B, D, C, A, B, A \rangle$ (different)
- $Z = \langle B, C, A \rangle$ is a common subsequence
- $Z = \langle B, C, B, A \rangle$ is a longest common subsequence

Longest Common Subsequence

Longest Common Subsequence (LCS) value problem
input: sequences $X = \langle x_1, x_2, \ldots, x_m \rangle$ and $Y = \langle y_1, y_2, \ldots, y_n \rangle$
output: the length of a longest common subsequence of X and Y

Longest Common Subsequence (LCS) solution problem
input: (same)
output: a longest common subsequence of X and Y

Step 1: Characterize a baseline solution

- **Idea:** brute force
- Enumerate every subsequence Z of X; check if each Z is also a subsequence of Y; keep the maximum-length acceptable Z
- $\Theta((m+n)2^m)$ time
- Exponential; too slow

Step 1: Characterize a a solution recursively

- Recall input: $X = \langle x_1, x_2, \ldots, x_m \rangle$ and $Y = \langle y_1, y_2, \ldots, y_n \rangle$
- Recall *prefix:* X_i is first i elements of X
- We want to compute sequence $LCS(X, Y)$; need to define this recursively
- **Idea:** If last symbols $x_m = y_n$ match, then extend a shorter common subsequence: $LCS(X, Y) = LCS(X_{m-1}, Y_{n-1}) + \langle x_m \rangle$
- Else ($x_m \neq y_n$), have to omit x_m or y_n (or both)
    - Omit x_m (or both): $LCS(X, Y) = LCS(X_{m-1}, Y)$
    - Omit y_n (or both): $LCS(X, Y) = LCS(X, Y_{n-1})$
    - Want **longest** so $LCS(X, Y) =$ longer of $LCS(X_{m-1}, Y)$ and $LCS(X, Y_{n-1})$

Example

- Suppose $X = \langle A, B, A, D \rangle$ and $Y = \langle B, B, A, C, D \rangle$
- Last symbols match, $x_4 = y_5 = D$, so

$$\begin{aligned} LCS(X, Y) &= LCS(X_{m-1}, Y_{n-1}) \\ &= LCS(\langle A, B, A \rangle, \langle B, B, A, C \rangle) + \langle D \rangle \end{aligned}$$

- Now suppose $X = \langle A, B, A, C \rangle$ and Y is the same
- Last symbols differ, $x_4 = C$ but $y_5 = D$, so

$$\begin{aligned} LCS(X, Y) &= \text{ longer of } LCS(X_{m-1}, Y) \text{ and } LCS(X, Y_{n-1}) \\ &= \text{longer of } LCS(\langle A, B, A \rangle, Y) \text{ and } LCS(X, \langle B, B, A, C \rangle) \end{aligned}$$

Step 2: Recursive solution

```
function LCS-LENGTH(X[0..m], Y[0..n])
    if  m == 0 or n == 0  then
        return 0
    end if
    if  x_m == y_n  then
        return LCS − LENGTH(X[0..m − 1], Y[0..n − 1]) + 1
    end if
    a = LCS − LENGTH(X[0..m − 1], Y)
    b = LCS − LENGTH(X, Y[0..n − 1])
    return max(a, b)
end function
```

Step 2: Recursive table definition

- (Switching to **value** problem — find length not subsequence)
- Create table (2D array) $c[i][j]$ with invariant

$$c[i][j] = \text{length of a LCS of } X_i \text{ and } Y_j$$

- Base cases: if $i = 0$ or $j = 0$, $c[i][j] = 0$
- General cases:
  - if $x_i = y_j$: $c[i][j] = c[i-1][j-1] + 1$
  - else, $c[i][j] = \max(c[i][j-1], c[i-1][j])$

Step 3: Pseudocode for **Bottom-Up** Dyn. Prog. Alg.

```
function LCS-LENGTH(X[0..m], Y[0..n])
    Create new 2D array c[0..m][0..n]
    for i = 0 to m : c[i][0] = 0
    for j = 1 to n : c[0][j] = 0
    for i = 1 to m do
        for j = 1 to n do
            if x_i == y_j then
                c[i][j] = c[i − 1][j − 1] + 1
            else
                c[i][j] = max(c[i][j − 1], c[i − 1][j])
            end if
        end for
    end for
    return c[m][n]
end function
```

Analysis

- 2D array $c : \Theta(mn)$ space
- Base cases $j = 0 : \Theta(m)$ time
- Base cases $i = 0 : \Theta(n)$ time
- General cases: $\Theta(mn)$ time
- Total: $\Theta(m + n + mn) = \Theta(mn)$ time
- **Huge speedup:** $\Theta((m + n)2^m) \longrightarrow \Theta(mn)$
- Exponential $\longrightarrow$ polynomial

Improving Space Complexity

- **Observe:** in the general-case loop, the only row indices used are i and $i-1$
  - Either $c[i][j] = c[i-1][j-1] + 1$
  - Or $c[i][j] = \max(c[i][j-1], c[i-1][j])$
- $\therefore$ we only need two rows at a time
  - 1) the current row i we are initializing
  - 2) the previous row $i-1$
- Keep a **window** of only two rows
- This idea applies to many dynamic programming algorithms

Space-Efficient Bottom-Up LCS

```
function LCS-LENGTH(X[0..m], Y[0..n])
    Create arrays prev[0..n] and now[0..n], both filled with zeroes
    for i = 1 to m do
        swap(prev, now)
        for j = 1 to n do
            if x_i == y_j then
                now[j] = prev[j − 1] + 1
            else
                now[j] = max(now[j − 1], prev[j])
            end if
        end for
    end for
    return now[n]
end function
```

Space-Efficient Analysis

- Space:
  - Each array is $\Theta(n)$ space
  - Total $\Theta(n)$ space
  - (Better than $\Theta(mn)$)
- Time:
  - Initialize arrays: $\Theta(n)$
  - General case loops: $\Theta(mn)$ time
  - Total $\Theta(mn)$ time
  - (Same as previous version)