13. Dynamic Programming for Longest Common Subsequence

CPSC 535

Kevin A. Wortman

CALIFORNIA STATE UNIVERSITY FULLERTON

Big Idea: Alternative Kinds of Solutions

- So far
  - Step 2. Derive a **recurrence** for an optimal value.
  - Recall rod cutting:

$$r_i = \max_{1 \leq i \leq n} (p_i + r_{n-i})$$

  - Recall matrix chain multiplication:

$$r_{i,j} = \min_{1 \leq k \leq j} r_{i,k} + r_{k+1,j} + p_i p_k p_j$$

- Now: longest common subsequence (LCS)
  - not simply minimizing/maximizing one expression
  - instead, choose between **three alternatives**
  - **2D table**, like matrix chain

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: $X_k = \langle x_1, \ldots, x_k \rangle; X_0 = \langle \rangle$
  - if $X = \langle 2, 7, 8, 1, 7, 1, 2 \rangle$ then $X_3 = \langle 2, 7, 8 \rangle$
- 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

Design Process

1. Identify the problem's **solution** and **value**, and note which is our **goal**.
2. Derive a **recurrence** for an optimal value.
3. Design a divide-and-conquer algorithm that computes an **optimal value**.
4. Design a dynamic programming algorithm that computes an **optimal value**.
   4.1 **top-down** alternative: add table base case (**memoization**)
   4.2 **bottom-up** alternative: rewrite to use bottom-up loops instead of recursion
5. (if goal is a solution algo.) Design a dynamic programming algorithm that computes an **optimal solution**.

Longest Common Subsequence Step 1

1. Identify the problem's **solution** and **value**, and note which is our **goal**.

- **solution:** a sequence e.g. $\langle B, C, B, A \rangle$
- **value:** integer length of a sequence e.g. 4
- eventual goal is solution
- start with value

Longest Common Subsequence Step 2

2. Derive a **recurrence** for an optimal value.

- 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

Longest Common Subsequence Step 2

2. Derive a **recurrence** for an optimal value.

- **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

$$LCS(X, Y) = \text{ longer of } LCS(X_{m-1}, Y) \text{ and } LCS(X, Y_{n-1}))$$

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 $Y = \langle B, B, A, C, D \rangle$
- Last symbols differ, $x_4 = C$ but $y_5 = D$, so

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

Longest Common Subsequence Step 2

2. Derive a **recurrence** for an optimal value.

$$LCS(X_m, Y_n) = \begin{cases} 0 & m = 0 \\ 0 & n = 0 \\ LCS(X_{m-1}, Y_{n-1}) + 1 & x_m = x_n \\ \max(LCS(X_{m-1}, Y_n), LCS(X_m, Y_{n-1})) & \text{otherwise} \end{cases}$$

Longest Common Subsequence Step 3

3. Design a divide-and-conquer algorithm that computes an **optimal value**.

```
function LCS-DC(X[1..m], Y[1..n])
    if  m == 0 or n == 0  then
        return  0
    else if  X[m] == Y[n]  then
        return  LCS-DC(X[1..m − 1], Y[1..n − 1]) + 1
    else
        return  max(LCS-DC(X[1..m − 1], Y[1..n]), LCS-DC(X[1..m], Y[1..n − 1])
    end if
end function
```

Matrix Chain Multiplication Step 4.a

4. Design a dynamic programming algorithm that computes an **optimal value**.
   4.1 **top-down** alternative: add table base case (**memoization**)

- Recall **memoization:** use a hash dictionary to make a "memo" of pre-calculated solutions
- create hash table T
- use pair (m, n) as key in table T, storing $LCS(X_m, Y_n)$

Matrix Chain Multiplication Step 4.a

```
function LCS-MEMOIZED(X[1..m], Y[1..n])
    HASH-TABLE-CREATE(T)
    return LCS-M(T, X, Y)
end function
function LCS-M(T, X[1..m], Y[1..n])
    q = HASH-TABLE-SEARCH(T, (m, n))
    if q ≠ NIL then
        return q
    end if
    if m == 0 or n == 0 then
        q = 0
    else if X[m] == Y[n] then
        q = LCS-M(T, X[1..m − 1], Y[1..m − 1]) + 1
    else
        q = max(LCS-M(X[1..m − 1], Y[1..n]), LCS-M(X[1..m], Y[1..n − 1]))
    end if
    q.key = (m, n)
    HASH-TABLE-INSERT(q)
    return q
end function
```

Memoized Algorithm Analysis

- T contains $\Theta(n^2)$ pairs (m, n)
- each entry is inserted exactly once
- in the general case, LCS-M takes $\Theta(1)$ expected time
- $\Rightarrow$ LCS-MEMOIZED takes $\Theta(n^2)$ expected time

Longest Common Subsequence Step 4.b

4. Design a dynamic programming algorithm that computes an **optimal value**.
   4.1 **top-down** alternative: add table base case (**memoization**)
   4.2 **bottom-up** alternative: rewrite to use bottom-up loops instead of recursion

- create 2D array c where $c[i][j] = LCS(X_i, Y_j)$
- **bottom-up:** write an explicit **for** loop that computes and stores every general case
- need to order loops so we never use an uninitialized element
- ∴ initialize all base cases before any general case

Longest Common Subsequence Step 4.b

```
function LCS-BU(X[1..m], Y[1..n])
    Create array c[0..m][0..n]                      ▷ unusual index range
    for i from 0 to m do
        c[i][0] = 0
    end for
    for j from 1 to n do                            ▷ only initialize c[0][0] once
        c[0][j] = 0
    end for
    for i from 1 to m do
        for j from 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 − 1][j], c[i][j − 1])
            end if
        end for
    end for
    return c[m][n]
end function
```

Bottom-Up Analysis

- LCS-BU is clearly $\Theta(n^2)$ time
- (easy analysis)

Longest Common Subsequence Step 5

5. (if goal is a solution algo.) Design a dynamic programming algorithm that computes an **optimal solution**.

- **idea:** for each (i, j), record which alternative sub-solution defines $c[i][j]$:
  - $\nwarrow \equiv c[i-1][j-1]$
  - $\uparrow \equiv c[i-1][j]$
  - $\leftarrow \equiv c[i][j-1]$
- define
$$b[i][j] \in \{\nwarrow, \uparrow, \leftarrow\}$$
- rewrite $\max(c[i-1][j], c[i][j-1])$ as **if**/**else** so we can update $b[i][j]$

Longest Common Subsequence Step 5

```
function LCS-SOLUTION(X[1..m], Y[1..n])
    Create arrays c[0..m][0..n] and b[1..m][1..n]        ▷ different index ranges
    for i from 0 to m do
        c[i][0] = 0
    end for
    for j from 1 to n do                                  ▷ only initialize c[0][0] once
        c[0][j] = 0
    end for
    for i from 1 to m do
        for j from 1 to n do
            if X[i] == Y[j] then
                c[i][j] = c[i − 1][j − 1] + 1
                b[i][j] =↖
            else if c[i − 1][j] ≥ c[i][j − 1] then
                c[i][j] = c[i − 1][j]
                b[i][j] =↑
            else
                c[i][j] = c[i][j − 1]
                b[i][j] =←
            end if
        end for
    end for
    return LCS-BTRACK(b, X, i, j)
end function
```

Longest Common Subsequence Step 5

```
function LCS-BTRACK(b[1..m][1..n], X[1..m], i, j)
    if i == 0 or j == 0 then
        return ⟨⟩                                        ▷ empty sequence
    end if
    if b[i][j] == ↖ then
        return LCS-BTRACK(b, X, i − 1, j − 1) + ⟨x_i⟩   ▷ append
    else if b[i][j] == ↑ then
        return LCS-BTRACK(b, X, i − 1, j)
    else
        return LCS-BTRACK(b, X, i, j − 1)
```